9

MINIMUM COST FLOWS: BASIC ALGORITHMS

. . . men must walk, at least, before they dance.
—Alexander Pope

Chapter Outline

9.1 INTRODUCTION

The minimum cost flow problem is the central object of study in this book. In the last five chapters, we have considered two special cases of this problem: the shortest path problem and the maximum flow problem. Our discussion has been multifaceted: (1) We have seen how these problems arise in application settings as diverse as equipment replacement, project planning, production scheduling, census rounding, and analyzing round-robin tournaments; (2) we have developed a number of algorithmic approaches for solving these problems and studied their computational complexity; and (3) we have shown connections between these problems and more general problems in combinatorial optimization such as the minimum cut problem and a variety of min-max duality results. As we have seen, it is easy to understand the basic nature of shortest path and maximum flow problems and to develop core algorithms for solving them; nevertheless, designing and analyzing efficient algorithms is a very challenging task, requiring considerable ingenuity and considerable insight concerning both basic algorithmic strategies and their implementations.

As we begin to study more general minimum cost flow problems, we might ask ourselves a number of questions.

1. How much more difficult is it to solve the minimum cost flow problem than its shortest path and maximum flow specializations?
2. Can we use some of the same basic algorithmic strategies, such as label-setting and label-correcting methods, and the many variants of the augmenting path methods (e.g., shortest augmenting paths, scaling methods) for solving minimum cost flow problems?
3. The shortest path problem and the maximum flow problem address different components of the overall minimum cost flow problem: Shortest path problems consider arc flow costs but no flow capacities; maximum flow problems consider capacities but only the simplest cost structure. Since the minimum cost flow problem combines these problem ingredients, can we somehow combine the material that we have examined for shortest path and maximum flow problems to develop optimality conditions, algorithms, and underlying theory for the minimum cost flow problem?

In this and the next two chapters, we provide (partial) answers to these questions. We develop a number of algorithms for solving the minimum cost flow problem. Although these algorithms are not as efficient as those for the shortest path and maximum flow problems, they still are quite efficient, and indeed, are among the most efficient algorithms known in applied mathematics, computer science, and operations research for solving large-scale optimization problems.

We also show that we can develop considerable insight and useful tools and methods of analysis by drawing on the material that we have developed already. For example, in order to give us a firm foundation for developing algorithms for solving minimum cost flow problems, in Section 9.3 we establish optimality conditions for minimum cost flow problems based on the notion of node potentials associated with the nodes in the underlying network. These node potentials are generalizations of the concept of distance labels that we used in our study of shortest path problems. Recall that we were able to use distance labels to characterize optimal shortest paths; in addition, we used the distance label optimality conditions as a starting point for developing the basic iterative label-setting and label-correcting algorithms for solving shortest path problems. We use the node potential in a similar fashion for minimum cost flow problems. The connection with shortest paths is much deeper, however, than this simple analogy between node potentials and distance labels. For example, we show how to interpret and find the optimal node potentials for a minimum cost flow problem by solving an appropriate shortest path problem: The optimal node potentials are equal to the negative of the optimal distance labels from this shortest path problem.

In addition, many algorithms for solving the minimum cost flow problem combine ingredients of both shortest path and maximum flow algorithms. Many of these algorithms solve a sequence of shortest path problems with respect to maximum flow-like residual networks and augmenting paths. (Actually, to define the residual network, we consider both cost and capacity considerations.) We consider four such algorithms in this chapter. The *cycle-canceling algorithm* uses shortest path computations to find augmenting cycles with negative flow costs; it then augments flows along these cycles and iteratively repeats these computations for detecting negative

cost cycles and augmenting flows. The *successive shortest path algorithm* incrementally loads flow on the network from some source node to some sink node, each time selecting an appropriately defined shortest path. The *primal-dual* and *out-of-kilter algorithms* use a similar algorithmic strategy: at every iteration, they solve a shortest path problem and augment flow along one or more shortest paths. They vary, however, in their tactics. The primal–dual algorithm uses a maximum flow computation to augment flow simultaneously along several shortest paths. Unlike all the other algorithms, the out-of-kilter algorithm permits arc flows to violate their flow bounds. It uses shortest path computations to find flows that satisfy both the flow bounds and the cost and capacity based optimality conditions.

The fact that we can implement iterative shortest path algorithms in so many ways demonstrates the versatility that we have in solving minimum cost flow problems. Indeed, as we shall see in the next two chapters, we have even more versatility. Each of the algorithms that we discuss in this chapter is pseudopolynomial for problems with integer data. As we shall see in Chapter 10, by using ideas such as scaling of the problem data, we can also develop polynomial-time algorithms.

Since minimum cost flow problems are linear programs, it is not surprising to discover that we can also use linear programming methodologies to solve minimum cost flow problems. Indeed, many of the various optimality conditions that we have introduced in previous chapters and that we consider in this chapter are special cases of the more general optimality conditions of linear programming. Moreover, we can interpret many of these results in the context of a general theory of duality for linear programs. In this chapter we develop these duality results for minimum cost flow problems. In Chapter 11 we study the application of the key algorithmic approach from linear programming, the simplex method, for the minimum cost flow problem. In this chapter we consider one other algorithm, known as the *relaxation algorithm*, for solving the minimum cost flow problem.

To begin our discussion of the minimum cost flow problem, we first consider some additional applications, which help to show the importance of this problem in practice. Before doing so, however, let us set our notation and some underlying definitions that we use throughout our discussion.

Notation and Assumptions

Let $G = (N, A)$ be a directed network with a *cost* c_{ij} and a *capacity* u_{ij} associated with every arc $(i, j) \in A$. We associate with each node $i \in N$ a number $b(i)$ which indicates its supply or demand depending on whether $b(i) > 0$ or $b(i) < 0$. The minimum cost flow problem can be stated as follows:

$$\text{Minimize} \quad z(x) = \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{9.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i) \qquad \text{for all } i \in N, \tag{9.1b}$$

$$0 \le x_{ij} \le u_{ij} \qquad \text{for all } (i, j) \in A. \tag{9.1c}$$

Let C denote the largest magnitude of any arc cost. Further, let U denote the

largest magnitude of any supply/demand or finite arc capacity. We assume that the lower bounds l_{ij} on arc flows are all zero. We further make the following assumptions:

Assumption 9.1. *All data (cost, supply/demand, and capacity) are integral.*

As noted previously, this assumption is not really restrictive in practice because computers work with rational numbers which we can convert to integer numbers by multiplying by a suitably large number.

Assumption 9.2. *The network is directed.*

We have shown in Section 2.4 that we can always fulfill this assumption by transforming any undirected network into a directed network.

Assumption 9.3. *The supplies/demands at the nodes satisfy the condition* $\sum_{i \in N} b(i) = 0$ *and the minimum cost flow problem has a feasible solution.*

We can determine whether the minimum cost flow problem has a feasible solution by solving a maximum flow problem as follows. Introduce a source node s^* and a sink node t^*. For each node i with $b(i) > 0$, add a "source" arc (s^*, i) with capacity $b(i)$, and for each node i with $b(i) < 0$, add a "sink" arc (i, t^*) with capacity $-b(i)$. Now solve a maximum flow problem from s^* to t^*. If the maximum flow saturates all the source arcs, the minimum cost flow problem is feasible; otherwise, it is infeasible. For the justification of this method, see Application 6.1 in Section 6.2.

Assumption 9.4. *We assume that the network G contains an uncapacitated directed path (i.e., each arc in the path has infinite capacity) between every pair of nodes.*

We impose this condition, if necessary, by adding *artificial* arcs $(1, j)$ and $(j, 1)$ for each $j \in N$ and assigning a large cost and infinite capacity to each of these arcs. No such arc would appear in a minimum cost solution unless the problem contains no feasible solution without artificial arcs.

Assumption 9.5. *All arc costs are nonnegative.*

This assumption imposes no loss of generality since the arc reversal transformation described in Section 2.4 converts a minimum cost flow problem with negative arc lengths to those with nonnegative arc lengths. This transformation, however, requires that all arcs have finite capacities. When some arcs are uncapacitated, we assume that the network contains no directed negative cost cycle of infinite capacity. If the network contains any such cycles, the optimal value of the minimum cost flow problem is unbounded; moreover, we can detect such a situation by using the search algorithm described in Section 3.4. In the absence of a negative cycle with infinite capacity, we can make each uncapacitated arc capacitated by setting its capacity equal to B, where B is the sum of all arc capacities and the supplies of all supply nodes; we justify this transformation in Exercise 9.36.

Residual Network

Our algorithms rely on the concept of residual networks. The residual network $G(x)$ corresponding to a flow x is defined as follows. We replace each arc $(i, j) \in A$ by two arcs (i, j) and (j, i). The arc (i, j) has cost c_{ij} and *residual capacity* $r_{ij} = u_{ij} - x_{ij}$, and the arc (j, i) has cost $c_{ji} = -c_{ij}$ and residual capacity $r_{ji} = x_{ij}$. The residual network consists *only* of arcs with positive residual capacity.

9.2 APPLICATIONS

Minimum cost flow problems arise in almost all industries, including agriculture, communications, defense, education, energy, health care, manufacturing, medicine, retailing, and transportation. Indeed, minimum cost flow problems are pervasive in practice. In this section, by considering a few selected applications that arise in distribution systems planning, medical diagnosis, public policy, transportation, manufacturing, capacity planning, and human resource management, we give a passing glimpse of these applications. This discussion is intended merely to introduce several important applications and to illustrate some of the possible uses of minimum cost flow problems in practice. Taken together, the exercises in this chapter and in Chapter 11 and the problem descriptions in Chapter 19 give a much more complete picture of the full range of applications of minimum cost flows.

Application 9.1 Distribution Problems

A large class of network flow problems centers around shipping and distribution applications. One core model might be best described in terms of shipments from plants to warehouses (or, alternatively, from warehouses to retailers). Suppose that a firm has p plants with known supplies and q warehouses with known demands. It wishes to identify a flow that satisfies the demands at the warehouses from the available supplies at the plants and that minimizes its shipping costs. This problem is a well-known special case of the minimum cost flow problem, known as the *transportation problem*. We next describe in more detail a slight generalization of this model that also incorporates manufacturing costs at the plants.

A car manufacturer has several manufacturing plants and produces several car models at each plant that it then ships to geographically dispersed retail centers throughout the country. Each retail center requests a specific number of cars of each model. The firm must determine the production plan of each model at each plant and a shipping pattern that satisfies the demands of each retail center and minimizes the overall cost of production and transportation.

We describe this formulation through an example. Figure 9.1 illustrates a situation with two manufacturing plants, two retailers, and three car models. This model has four types of nodes: (1) *plant nodes*, representing various plants; (2) *plant/model nodes*, corresponding to each model made at a plant; (3) *retailer/model nodes*, corresponding to the models required by each retailer; and (4) *retailer nodes* corresponding to each retailer. The network contains three types of arcs.

1. *Production arcs*. These arcs connect a plant node to a plant/model node; the cost of this arc is the cost of producing the model at that plant. We might place

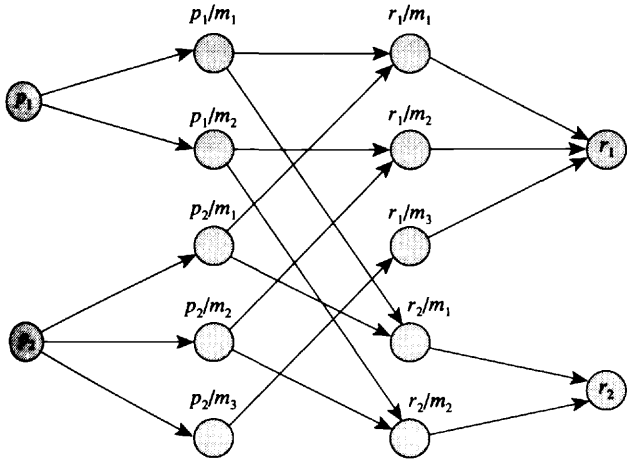


Figure 9.1 Production-distribution model.

lower and upper bounds on these arcs to control for the minimum and maximum production of each particular car model at the plants.

2. *Transportation arcs*. These arcs connect plant/model nodes to retailer/model nodes; the cost of such an arc is the total cost of shipping one car from the manufacturing plant to the retail center. Any such arc might correspond to a complex distribution channel with, for example, three legs: (a) a delivery from a plant (by truck) to a rail system; (b) a delivery from the rail station to another rail station elsewhere in the system; and (c) a delivery from the rail station to a retailer (by a local delivery truck). The transportation arcs might have lower or upper bounds imposed on their flows to model contractual agreements with shippers or capacities imposed on any distribution channel.
3. *Demand arcs*. These arcs connect retailer/model nodes to the retailer nodes. These arcs have zero costs and positive lower bounds which equal the demand of that model at that retail center.

Clearly, the production and shipping schedules for the automobile company correspond in a one-to-one fashion with the feasible flows in this network model. Consequently, a minimum cost flow would yield an optimal production and shipping schedule.

Application 9.2 Reconstructing the Left Ventricle from X-ray Projections

This application describes a network flow model for reconstructing the three-dimensional shape of the left ventricle from biplane angiocardiograms that the medical profession uses to diagnose heart diseases. To conduct this analysis, we first reduce the three-dimensional reconstruction problem into several two-dimensional problems by dividing the ventricle into a stack of parallel cross sections. Each two-dimensional cross section consists of one connected region of the left ventricle.

During a cardiac catheterization, doctors inject a dye known as Roentgen contrast agent into the ventricle; by taking x-rays of the dye, they would like to determine what portion of the left ventricle is functioning properly (i.e., permitting the flow of blood). Conventional biplane x-ray installations do not permit doctors to obtain a complete picture of the left ventricle; rather, these x-rays provide one-dimensional projections that record the total intensity of the dye along two axes (see Figure 9.2). The problem is to determine the distribution of the cloud of dye within the left ventricle and thus the shape of the functioning portion of the ventricle, assuming that the dye mixes completely with the blood and fills the portions that are functioning properly.

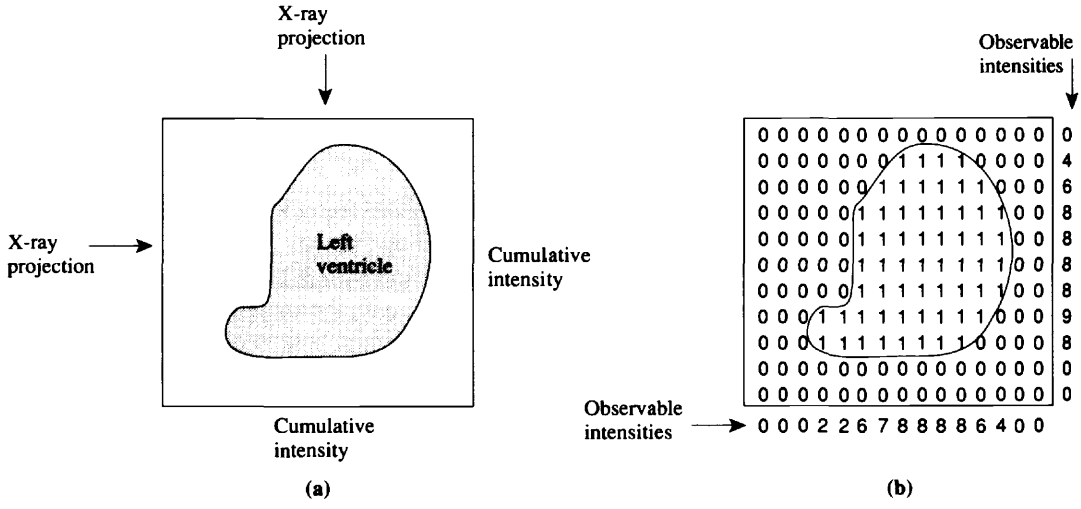


Figure 9.2 Using x-ray projections to measure a left ventricle.

We can conceive of a cross section of the ventricle as a $p \times r$ binary matrix: a 1 in a position indicates that the corresponding segment allows blood to flow and a 0 indicates that it does not permit blood to flow. The angiocardiograms give the cumulative intensity of the contrast agent in two planes which we can translate into row and column sums of the binary matrix. The problem is then to construct the binary matrix given its row and column sums. This problem is a special case of the feasible flow problem that we discussed in Section 6.2.

Typically, the number of feasible solutions for such problems are quite large; and these solutions might differ substantially. To constrain the feasible solutions, we might use certain facts from our experience that indicate that a solution is more likely to contain certain segments rather than others. Alternatively, we can use *a priori* information: for example, after some small time interval, the cross sections might resemble cross sections determined in a previous examination. Consequently, we might attach a probability p_{ij} that a solution will contain an element (i, j) of the binary matrix and might want to find a feasible solution with the largest possible cumulative probability. This problem is equivalent to a minimum cost flow problem.

Application 9.3 Racial Balancing of Schools

In Application 1.10 in Section 1.3 we formulated the racial balancing of schools as a multicommodity flow problem. We now consider a related, yet important situation: seeking a racial balance of two ethnic communities (blacks and whites). In this case we show how to formulate the problem as a minimum cost flow problem.

As in Application 1.10, suppose that a school district has S schools. For the purpose of this formulation, we divide the school district into L district locations and let b_i and w_i denote the number of black and white students at location i. These locations might, for example, be census tracts, bus stops, or city blocks. The only restrictions on the locations is that they be finite in number and that there be a single distance measure d_{ij} that reasonably approximates the distance any student at location i must travel if he or she is assigned to school j. We make the reasonable assumption that we can compute the distances d_{ij} before assigning students to schools. School j can enroll u_j students. Finally, let $\underline{p}$ denote a lower bound and $\bar{p}$ denote an upper bound on the percentage of black students assigned to each school (we choose these numbers so that school j has same percentage of blacks as does the school district). The objective is to assign students to schools in a manner that maintains the stated racial balance and minimizes the total distance traveled by the students.

We model this problem as a minimum cost flow problem. Figure 9.3 shows the minimum cost flow network for a three-location, two-school problem. Rather than describe the general model formally, we merely describe the model ingredients for this figure. In this formulation we model each location i as two nodes l'_i and l''_i and each school j as two nodes s'_j and s''_j. The decision variables for this problem are

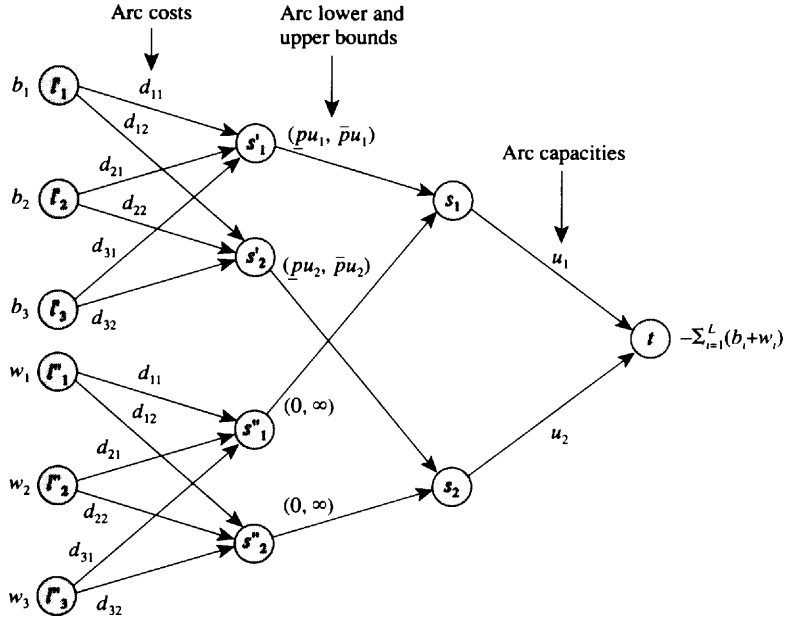


Figure 9.3 Network for the racial balancing of schools.

the number of black students assigned from location i to school j (which we represent by an arc from node l_i' to node s_j') and the number of white students assigned from location i to school j (which we represent by an arc from node l_i'' to node s_j''). These arcs are uncapacitated and we set their per unit flow cost equal to d_{ij}. For each j, we connect the nodes s_j' and s_j'' to the school node s_j. The flow on the arcs (s_j', s_j) and (s_j'', s_j) denotes the total number of black and white students assigned to school j. Since each school must satisfy lower and upper bounds on the number of black students it enrolls, we set the lower and upper bounds of the arc (s_j', s_j) equal to $(\underline{p}u_j, \bar{p}u_j)$. Finally, we must satisfy the constraint that school j enrolls at most u_j students. We incorporate this constraint in the model by introducing a sink node t and joining each school node j to node t by an arc of capacity u_j. As is easy to verify, this minimum cost flow problem correctly models the racial balancing application.

Application 9.4 Optimal Loading of a Hopping Airplane

A small commuter airline uses a plane, with a capacity to carry at most p passengers, on a "hopping flight," as shown in Figure 9.4(a). The hopping flight visits the cities $1, 2, 3, \ldots, n$, in a fixed sequence. The plane can pick up passengers at any node and drop them off at any other node. Let b_{ij} denote the number of passengers available at node i who want to go to node j, and let f_{ij} denote the fare per passenger from node i to node j. The airline would like to determine the number of passengers that the plane should carry between the various origins to destinations in order to maximize the total fare per trip while never exceeding the plane capacity.

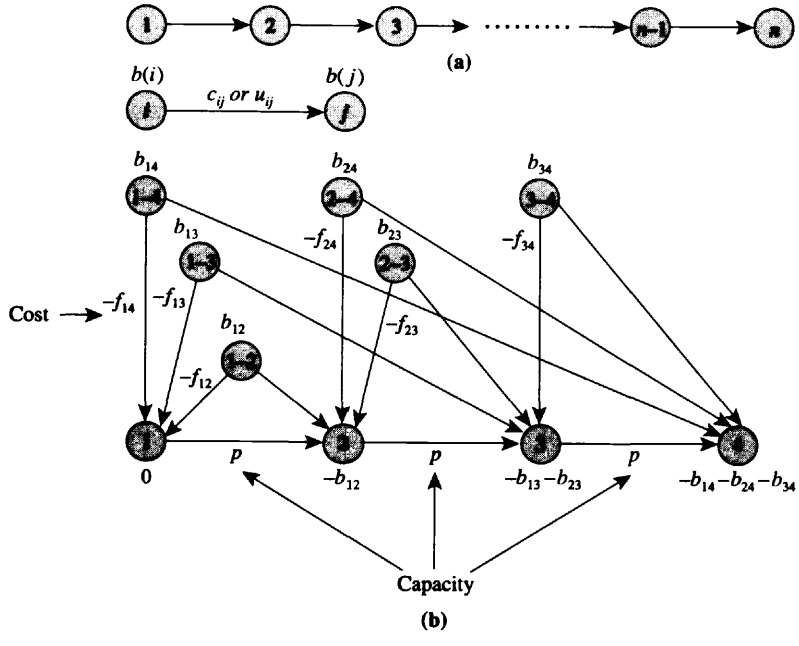


Figure 9.4 Formulating the hopping plane flight problem as a minimum cost flow problem.

Figure 9.4(b) shows a minimum cost flow formulation of this hopping plane flight problem. The network contains data for only those arcs with nonzero costs and with finite capacities: Any arc without an associated cost has a zero cost; any arc without an associated capacity has an infinite capacity. Consider, for example, node 1. Three types of passengers are available at node 1, those whose destination is node 2, node 3, or node 4. We represent these three types of passengers by the nodes 1–2, 1–3, and 1–4 with supplies b_{12}, b_{13}, and b_{14}. A passenger available at any such node, say 1–3, either boards the plane at its origin node by flowing through the arc (1–3, 1), and thus incurring a cost of $-f_{13}$ units, or never boards the plane which we represent by the flow through the arc (1–3, 3). In Exercise 9.13 we ask the reader to show that this formulation correctly models the hopping plane application.

Application 9.5 Scheduling with Deferral Costs

In some scheduling applications, jobs do not have any fixed completion times, but instead incur a deferral cost for delaying their completion. Some of these scheduling problems have the following characteristics: one of q identical processors (machines) needs to process each of p jobs. Each job j has a fixed processing time α_j that does not depend on which machine processes the job, or which jobs precede or follow the job. Job j also has a *deferral cost* $c_j(\tau)$, which we assume is a monotonically nondecreasing function of τ, the completion time of the job. Figure 9.5(a) illustrates one such deferral cost function. We wish to find a schedule for the jobs, with completion times denoted by $\tau_1, \tau_2, \ldots, \tau_p$, that minimizes the total deferral cost $\sum_{j=1}^{p} c_j(\tau_j)$. This scheduling problem is difficult if the jobs have different processing times, but can be modeled as a minimum cost flow problem for situations with uniform processing times (i.e., $\alpha_j = \alpha$ for each $j = 1, \ldots, p$).

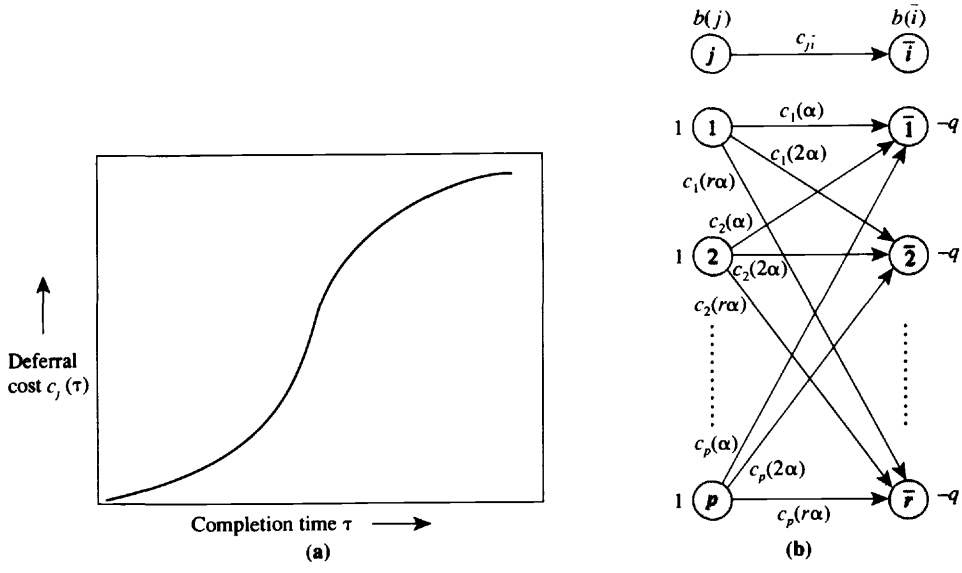


Figure 9.5 Formulating the scheduling problem with deferral costs.

Since the deferral costs are monotonically nondecreasing with time, in some optimal schedule the machines will process the jobs one immediately after another (i.e., the machines incur no *idle time*). As a consequence, in some optimal schedule the completion of each job will be $k\alpha$ for some constant k. The first job assigned to every machine will have a completion time of α units, the second job assigned to every machine will have a completion time of 2α units, and so on. This observation allows us to formulate the scheduling as a minimum cost flow problem in the network shown in Figure 9.5(b).

Assume, for simplicity, that $r = p/q$ is an integer. This assumption implies that we will assign exactly r jobs to each machine. (There is no loss of generality in imposing this assumption because we can add dummy jobs so that p/q becomes an integer.) The network has p job nodes, $1, 2, \ldots, p$, each with 1 unit of supply; it also has r position nodes, $\bar{1}, \bar{2}, \ldots, \bar{r}$, each with a demand of q units, indicating that the position has the capability to process q jobs. The flow on each arc $(j, \bar{i})$ is 1 or 0, depending on whether the schedule does or does not assign job j to the ith position of some machine. If we assign job j to the ith position on any machine, its completion time is $i\alpha$ and its deferral cost is $c_j(i\alpha)$. Therefore, arc $(j, \bar{i})$ has a cost of $c_j(i\alpha)$. Feasible schedules correspond, in a one-to-one fashion, with feasible flows in the network and both have the same cost. Consequently, a minimum cost flow will prescribe a schedule with the least possible deferral cost.

Application 9.6 Linear Programs with Consecutive 1's in Columns

Many linear programming problems of the form

$$\text{Minimize} \quad cx$$

subject to

$$\mathcal{A}x \geq b,$$

$$x \geq 0,$$

have a special structure that permits us to solve the problem more efficiently than general-purpose linear programs. Suppose that the $p \times q$ matrix constraint matrix $\mathcal{A}$ is a 0–1 matrix satisfying the property that all of the 1's in each column appear consecutively (i.e., with no intervening zeros). We show how to transform this problem into a minimum cost flow problem. We illustrate our transformation using the following linear programming example:

$$\text{Minimize} \quad cx \tag{9.2a}$$

subject to

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix} x \geq \begin{bmatrix} 5 \\ 12 \\ 10 \\ 6 \end{bmatrix}, \tag{9.2b}$$

$$x \geq 0. \tag{9.2c}$$

We first bring each constraint in (9.2b) into an equality form by introducing a "surplus" variable y_i for each row i in (9.2b). We then add a redundant row $0 \cdot x + 0 \cdot y = 0$ to the set of constraints. These changes produce the following equivalent formulation of the linear program:

$$\text{Minimize} \quad cx \tag{9.3a}$$

subject to

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & -1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \\ x \\ \\ y \\ \\ \end{bmatrix} = \begin{bmatrix} 5 \\ 12 \\ 10 \\ 6 \\ 0 \end{bmatrix}, \tag{9.3b}$$

$$x \geq 0. \tag{9.3c}$$

We next perform the following elementary row operation for each $i = p, p - 1, \ldots, 1$, in the stated order: We subtract the ith constraint in (9.3b) from the $(i + 1)$th constraint. These operations create the following equivalent linear program:

$$\text{Minimize} \quad cx \tag{9.4a}$$

subject to

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \\ x \\ \\ y \\ \\ \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ -2 \\ -4 \\ -6 \end{bmatrix}, \tag{9.4b}$$

$$x \geq 0. \tag{9.4c}$$

Notice that in this form the constraints (9.4b) clearly define the mass balance constraints of a minimum cost flow problem because each column contains one $+1$ and one -1. Also notice that the entries in the right-hand-side vector sum to zero, which is a necessary condition for feasibility. Figure 9.6 gives the minimum cost flow problem corresponding to this linear program.

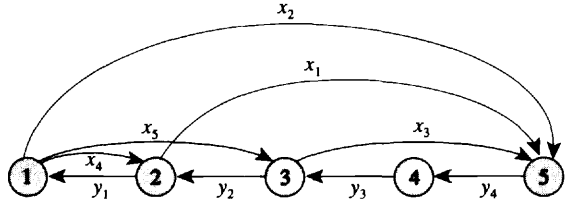


Figure 9.6 Formulating a linear program with consecutive ones as a minimum cost flow problem.

We have used a specific numerical example to illustrate the transformation of a linear program with consecutive 1's into a minimum cost flow problem. It is easy to show that this transformation is valid in general as well. For a linear program with p rows and q columns, the corresponding network has $p + 1$ nodes, one corresponding to each row, as well as one extra node that corresponds to an additional

"null row." Each column $\mathcal{A}_k$ in the linear program that has consecutive 1's in rows i to j becomes an arc $(i, j + 1)$ of cost c_k. Each surplus variable y_i becomes an arc $(i + 1, i)$ of zero cost. Finally, the supply/demand of a node i is $b(i) - b(i - 1)$.

Despite the fact that linear programs with consecutive 1's might appear to be very special, and even contrived, this class of problems arises in a surprising number of applications. We illustrate the range of applications with three practical examples. We leave the formulations of these applications as minimum cost flow problems as exercises to the reader.

Optimal capacity scheduling. A vice-president of logistics of a large manufacturing firm must contract for $d(i)$ units of warehousing capacity for the time periods $i = 1, 2, \ldots, n$. Let c_{ij} denote the cost of acquiring 1 unit of capacity at the beginning of period i, which is available for possible use throughout periods i, $i + 1, \ldots, j - 1$ (assume that we relinquish this warehousing capacity at the beginning of period j). The vice-president wants to know how much capacity to acquire, at what times, and for how many subsequent periods, to meet the firm's requirements at the lowest possible cost. This optimization problem arises because of possible savings that the firm might accrue by undertaking long-term leasing contracts at favorable times, even though these commitments might create excess capacity during some periods.

Employment scheduling. The vice-president of human resources of a large retail company must determine an employment policy that properly balances the cost of hiring, training, and releasing short-term employees, with the expense of having idle employees on the payroll for time periods when demand is low. Suppose that the company knows the minimum labor requirement d_j for each period $j = 1, \ldots, n$. Let c_{ij} denote the cost of hiring someone at the beginning of period i and releasing him at the end of period $j - 1$. The vice-president would like to identify an employment policy that meets the labor requirements and minimizes the cost of hiring, training, and releasing employees.

Equipment replacement. A job shop must periodically replace its capital equipment because of machine wear. As a machine ages, it breaks down more frequently and so becomes more expensive to operate. Furthermore, as a machine ages, its salvage value decreases. Let c_{ij} denote the cost of buying a particularly important machine at the beginning of period i, plus the cost of operating the machine over the periods $i, i + 1, \ldots, j - 1$, minus the salvage cost of the machine at the beginning of period j. The *equipment replacement problem* attempts to obtain a replacement plan that minimizes the total cost of buying, selling, and operating the machine over a planning horizon of n years, assuming that the job shop must have at least 1 unit of this machine in service at all times.

9.3 OPTIMALITY CONDITIONS

In our discussion of shortest path problems in Section 5.2, we saw that a set of distance labels $d(i)$ defines shortest path distances from a specified node s to every other node in the network if and only if they represent distances along some paths

from node s and satisfy the following shortest path optimality conditions:

$$d(j) \leq d(i) + c_{ij} \qquad \text{for all } (i, j) \in A. \tag{9.5}$$

These optimality conditions are useful in several respects. First, they give us a simple validity check to see whether a given set of distance labels does indeed define shortest paths. Similarly, the optimality conditions provide us with a method for determining whether or not a given set of paths, one from node s to every other node in the network, constitutes a set of shortest paths from node s. We simply compute the lengths of these paths and see if these distances satisfy the optimality conditions. In both cases, the optimality conditions provide us with a "certificate" of optimality, that is, an assurance that a set of distance labels or a set of paths is optimal. One nice feature of the certificate is its ease of use. We need not invoke any complex algorithm to certify that a solution is optimal; we simply check the optimality conditions. The optimality conditions are also valuable for other reasons; as we saw in Chapter 5, they can suggest algorithms for solving a shortest path problem: For example, the generic label-correcting algorithm uses the simple idea of repeatedly replacing $d(j)$ by $d(i) + c_{ij}$ if $d(j) > d(i) + c_{ij}$ for some arc (i, j). Finally, the optimality conditions provide us with a mechanism for establishing the validity of algorithms for the shortest path problem. To show that an algorithm correctly finds the desired shortest paths, we verify that the solutions they generate satisfy the optimality conditions.

These various uses of the shortest path optimality conditions suggest that similar sets of conditions might be valuable for designing and analyzing algorithms for the minimum cost flow problem. Accordingly, rather than launching immediately into a discussion of algorithms for solving the minimum cost flow problem, we first pause to describe a few different optimality conditions for this problem. All the optimality conditions that we state have an intuitive network interpretation and are rather direct extensions of their shortest path counterparts. We will consider three different (but equivalent) optimality conditions: (1) negative cycle optimality conditions, (2) reduced cost optimality conditions, and (3) complementary slackness optimality conditions.

Negative Cycle Optimality Conditions

The negative cycle optimality conditions stated next are a direct consequence of the flow decomposition property stated in Theorem 3.5 and our definition of residual networks given at the end of Section 9.1.

Theorem 9.1 (Negative Cycle Optimality Conditions). *A feasible solution x^* is an optimal solution of the minimum cost flow problem if and only if it satisfies the negative cycle optimality conditions: namely, the residual network $G(x^*)$ contains no negative cost (directed) cycle.*

Proof. Suppose that x is a feasible flow and that $G(x)$ contains a negative cycle. Then x cannot be an optimal flow, since by augmenting positive flow along the cycle we can improve the objective function value. Therefore, if x^* is an optimal flow, then $G(x^*)$ cannot contain a negative cycle. Now suppose that x^* is a feasible flow

and that $G(x^*)$ contains no negative cycle. Let x° be an optimal flow and $x^* \neq x^\circ$. The augmenting cycle property stated in Theorem 3.7 shows that we can decompose the difference vector $x^\circ - x^*$ into at most m augmenting cycles with respect to the flow x^* and the sum of the costs of flows on these cycles equals $cx^\circ - cx^*$. Since the lengths of all the cycles in $G(x^*)$ are nonnegative, $cx^\circ - cx^* \geq 0$, or $cx^\circ \geq cx^*$. Moreover, since x° is an optimal flow, $cx^\circ \leq cx^*$. Thus $cx^\circ = cx^*$, and x^* is also an optimal flow. This argument shows that if $G(x^*)$ contains no negative cycle, then x^* must be optimal, and this conclusion completes the proof of the theorem.◆

Reduced Cost Optimality Conditions

To develop our second and third optimality conditions, let us make one observation. First, note that we can write the shortest path optimality conditions in the following equivalent form:

$$c_{ij}^d = c_{ij} + d(i) - d(j) \geq 0 \qquad \text{for all arcs } (i, j) \in A. \tag{9.6}$$

This expression has the following interpretation: c_{ij}^d is an optimal "reduced cost" for arc (i, j) in the sense that it measures the cost of this arc relative to the shortest path distances $d(i)$ and $d(j)$. Notice that with respect to the optimal distances, every arc in the network has a nonnegative reduced cost. Moreover, since $d(j) = d(i) + c_{ij}$, if arc (i, j) is on a shortest path connecting the source node s to any other node, the shortest path uses only zero reduced cost arcs. Consequently, once we know the optimal distances, the problem is very easy to solve: We simply find a path from node s to every other node that uses only arcs with zero reduced costs. This interpretation raises a natural question: Is there a similar set of conditions for more general minimum cost flow problems?

Suppose that we associate a real number $\pi(i)$, unrestricted in sign, with each node $i \in N$. We refer to $\pi(i)$ as the *potential* of node i. We show in Section 9.4 that $\pi(i)$ is the linear programming dual variable corresponding to the mass balance constraint of node i. For a given set of node potentials π, we define the *reduced* cost of an arc (i, j) as $c_{ij}^\pi = c_{ij} - \pi(i) + \pi(j)$. These reduced costs are applicable to the residual network as well as the original network. We define the reduced costs in the residual network just as we did the costs, but now using c_{ij}^π in place of c_{ij}. The following properties will prove to be useful in our subsequent developments in this and later chapters.

Property 9.2

(a) *For any directed path P from node k to node l,* $\sum_{(i,j)\in P} c_{ij}^\pi = \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l)$.

(b) *For any directed cycle W,* $\sum_{(i,j)\in W} c_{ij}^\pi = \sum_{(i,j)\in W} c_{ij}$.

The proof of this property is similar to that of Property 2.5. Notice that this property implies that the node potentials do not change the shortest path between any pair of nodes k and l, since the potentials increase the length of every path by a constant amount $\pi(l) - \pi(k)$. This property also implies that if W is a negative cycle with respect to c_{ij} as arc costs, it is also a negative cycle with respect to c_{ij}^π

as arc costs. We can now provide an alternative form of the negative cycle optimality conditions, stated in terms of the reduced costs of the arcs.

Theorem 9.3 (Reduced Cost Optimality Conditions). *A feasible solution x^* is an optimal solution of the minimum cost flow problem if and only if some set of node potentials π satisfy the following reduced cost optimality conditions:*

$$c_{ij}^{\pi} \geq 0 \qquad \text{for every arc } (i, j) \text{ in } G(x^*). \tag{9.7}$$

Proof. We shall prove this result using Theorem 9.1. To show that the negative cycle optimality conditions is equivalent to the reduced cost optimality conditions, suppose that the solution x^* satisfies the latter conditions. Therefore, $\sum_{(i,j)\in W} c_{ij}^{\pi} \geq 0$ for every directed cycle W in $G(x^*)$. Consequently, by Property 9.2(b), $\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} c_{ij} \geq 0$, so $G(x^*)$ contains no negative cycle.

To show the converse, assume that for the solution x^*, $G(x^*)$ contains no negative cycle. Let $d(\cdot)$ denote the shortest path distances from node 1 to all other nodes in $G(x^*)$. Recall from Section 5.2 that if the network contains no negative cycle, the distance labels $d(\cdot)$ are well defined and satisfy the conditions $d(j) \leq d(i) + c_{ij}$ for all (i, j) in $G(x^*)$. We can restate these inequalities as $c_{ij} - (-d(i)) + (-d(j)) \geq 0$, or $c_{ij}^{\pi} \geq 0$ if we define $\pi = -d$. Consequently, the solution x^* satisfies the reduced cost optimality conditions. ◆

In the preceding theorem we characterized an optimal flow x as a flow that satisfied the conditions $c_{ij}^{\pi} \geq$ for all (i, j) in $G(x)$ for some set of node potentials π. In the same fashion, we could define "optimal node potentials" as a set of node potentials π that satisfy the conditions $c_{ij}^{\pi} \geq 0$ for all (i, j) in $G(x)$ for some feasible flow x.

We might note that the reduced cost optimality conditions have a convenient economic interpretation. Suppose that we interpret c_{ij} as the cost of transporting 1 unit of a commodity from node i to node j through the arc (i, j), and we interpret $\mu(i) \equiv -\pi(i)$ as the cost of obtaining a unit of this commodity at node i. Then $c_{ij} + \mu(i)$ is the cost of the commodity at node j if we obtain it at node i and transport it to node j. The reduced cost optimality condition, $c_{ij} - \pi(i) + \pi(j) \geq 0$, or equivalently, $\mu(j) \leq c_{ij} + \mu(i)$, states that the cost of obtaining the commodity at node j is no more than the cost of the commodity if we obtain it at node i and incur the transportation cost in sending it from node i to j. The cost at node j might be smaller than $c_{ij} + \mu(i)$ because there might be a more cost-effective way to transport the commodity to node j via other nodes.

Complementary Slackness Optimality Conditions

Both Theorems 9.1 and 9.3 provide means for establishing optimality of solutions to the minimum cost flow problem by formulating conditions imposed on the residual network; we shall now restate these conditions in terms of the original network.

Theorem 9.4 (Complementary Slackness Optimality Conditions). *A feasible solution x^* is an optimal solution of the minimum cost flow problem if and only if for some set of node potentials π, the reduced costs and flow values satisfy the following complementary slackness optimality conditions for every arc $(i, j) \in A$:*

$$\text{If } c_{ij}^{\pi} > 0, \text{ then } x_{ij}^{*} = 0. \tag{9.8a}$$

$$\text{If } 0 < x_{ij}^{*} < u_{ij}, \text{ then } c_{ij}^{\pi} = 0. \tag{9.8b}$$

$$\text{If } c_{ij}^{\pi} < 0, \text{ then } x_{ij}^{*} = u_{ij}. \tag{9.8c}$$

Proof. We show that the reduced cost optimality conditions are equivalent to (9.8). To establish this result, we first prove that if the node potentials π and the flow vector x satisfy the reduced cost optimality conditions, then they must satisfy (9.8). Consider three possibilities for any arc $(i, j) \in A$.

Case 1. If $c_{ij}^{\pi} > 0$, the residual network cannot contain the arc (j, i) because $c_{ji}^{\pi} = -c_{ij}^{\pi} < 0$ for that arc, contradicting (9.7). Therefore, $x_{ij}^{*} = 0$.

Case 2. If $0 < x_{ij}^{*} < u_{ij}$, the residual network contains both the arcs (i, j) and (j, i). The reduced cost optimality conditions imply that $c_{ij}^{\pi} \geq 0$ and $c_{ji}^{\pi} \geq 0$. But since $c_{ji}^{\pi} = -c_{ij}^{\pi}$, these inequalities imply that $c_{ij}^{\pi} = c_{ji}^{\pi} = 0$.

Case 3. If $c_{ij}^{\pi} < 0$, the residual network cannot contain the arc (i, j) because $c_{ij}^{\pi} < 0$ for that arc, contradicting (9.7). Therefore, $x_{ij}^{*} = u_{ij}$.

We have thus shown that if the node potentials π and the flow vector x satisfy the reduced cost optimality conditions, they also satisfy the complementary slackness optimality conditions. In Exercise 9.28 we ask the reader to prove the converse result: If the pair (x, π) satisfies the complementary slackness optimality conditions, it also satisfies the reduced cost optimality conditions. ◆

Those readers familiar with linear programming might notice that these conditions are the complementary slackness conditions for a linear programming problem whose variables have upper bounds; this association explains the choice of the name complementary slackness.

9.4 MINIMUM COST FLOW DUALITY

When we were introducing shortest path problems with nonnegative arc costs in Chapter 4, we considered a string model with knots representing the nodes of the network and with a string of length c_{ij} connecting the ith and jth knots. To solve the shortest path problem between a designated source node s and sink node t, we hold the string at the knots s and t and pull them as far apart as possible. As we noted in our previous discussion, if $d(i)$ denotes the distance from the source node s to node i along the shortest path and nodes i and j are any two nodes on this path, then $d(i) + c_{ij} \geq d(j)$. The shortest path distances might satisfy this inequality as a strict inequality if the string from node i to node j is not taut. In this string solution, since we are pulling the string apart as far as possible, we are obtaining the optimal shortest path distance between nodes s and t by solving a *maximization* problem. We could cast this problem formally as the following maximization problem:

$$\text{Maximize} \quad d(t) - d(s) \tag{9.9a}$$

subject to

$$d(j) - d(i) \leq c_{ij} \qquad \text{for all } (i, j) \in A. \tag{9.9b}$$

In this formulation, $d(s) = 0$. As we have noted in Chapter 4, if d is any vector of distance labels satisfying the constraints of this problem and the path P defined as $s - i_1 - i_2 - \cdots i_k - t$ is any path from node s to node t, then

$$d(i_1) - d(s) \leq c_{si_1}$$

$$d(i_2) - d(i_1) \leq c_{i_1 i_2}$$

$$\vdots$$

$$d(t) - d(i_k) \leq c_{i_k t},$$

so by adding these inequalities and using the fact that $d(s) = 0$, we see that

$$d(t) \leq c_{si_1} + c_{i_1 i_2} + \cdots + c_{i_k t}.$$

This result shows that if d is any feasible vector to the optimization problem (9.9), then $d(t)$ is a lower bound on the length of any path from node s to node t and therefore is a lower bound on the shortest distance between these nodes. As we see from the string solution, if we choose the distance labels $d(\cdot)$ appropriately (as the distances obtained from the string solution), $d(t)$ equals the shortest path distance.

This discussion shows the connection between the shortest path problem and a related maximization problem (9.9). In our discussion of the maximum flow problem, we saw a similar relationship, namely, the max-flow min-cut theorem, which tells us that associated with every maximum flow problem is an associated minimization problem. Moreover, since the maximum flow equals the minimum cut, the optimal value of these two associated problems is the same. These two results are special cases of a more general property that applies to any minimum cost flow problem, and that we now establish.

For every linear programming problem, which we subsequently refer to as a *primal* problem, we can associate another intimately related linear programming problem, called its *dual*. For example, the objective function value of *any* feasible solution of the dual is less than or equal to the objective function of any feasible solution of the primal. Furthermore, the maximum objective function value of the dual equals the minimum objective function of the primal. This duality theory is fundamental to an understanding of the theory of linear programming. In this section we state and prove these duality theory results for the minimum cost flow problem.

While forming the dual of a (primal) linear programming problem, we associate a *dual variable* with every constraint of the primal except for the nonnegativity restriction on arc flows. For the minimum cost flow problem stated in (9.1), we associate the variable $\pi(i)$ with the mass balance constraint of node i and the variable α_{ij} with the capacity constraint of arc (i, j). In terms of these variables, the *dual minimum cost flow problem* can be stated as follows:

$$\text{Maximize} \quad w(\pi, \alpha) = \sum_{i \in N} b(i)\pi(i) - \sum_{(i,j) \in A} u_{ij}\alpha_{ij} \tag{9.10a}$$

subject to

$$\pi(i) - \pi(j) - \alpha_{ij} \leq c_{ij} \qquad \text{for all } (i, j) \in A, \tag{9.10b}$$

$$\alpha_{ij} \geq 0 \qquad \text{for all } (i, j) \in A \quad \text{and} \quad \pi(j) \text{ unrestricted for all } j \in N. \tag{9.10c}$$

Note that the shortest path dual problem (9.9) is a special case of this model: For the shortest path problem, $b(s) = 1$, $b(t) = -1$, and $b(i) = 0$ otherwise. Also, since the shortest path problem contains no arc capacities, we can eliminate the α_{ij} variables. Therefore, if we let $d(i) = -\pi(i)$, the dual minimum cost flow problem (9.10) becomes the shortest path dual problem (9.9).

Our first duality result for the general minimum cost flow problem is known as the *weak duality theorem.*

Theorem 9.5 (Weak Duality Theorem). *Let $z(x)$ denote the objective function value of some feasible solution x of the minimum cost flow problem and let $w(\pi, \alpha)$ denote the objective function value of some feasible solution (π, α) of its dual. Then $w(\pi, \alpha) \leq z(x)$.*

Proof. We multiply both sides of (9.10b) by x_{ij} and sum these weighted inequalities for all $(i, j) \in A$, obtaining

$$\sum_{(i,j)\in A} (\pi(i) - \pi(j))x_{ij} - \sum_{(i,j)\in A} \alpha_{ij}x_{ij} \leq \sum_{(i,j)\in A} c_{ij}x_{ij}. \tag{9.11}$$

Notice that $cx - c^{\pi}x = \sum_{(i,j)\in A} (\pi(i) - \pi(j))x_{ij}$ [because $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$]. Next notice that Property 2.4 in Section 2.4 implies that $cx - c^{\pi}x$ equals $\sum_{i\in N} b(i)\pi(i)$. Therefore, the first term on the left-hand side of (9.11) equals $\sum_{i\in N} b(i)\pi(i)$. Next notice that replacing x_{ij} in the second term on the left-hand side of (9.11) by u_{ij} preserves the inequality because $x_{ij} \leq u_{ij}$ and $\alpha_{ij} \geq 0$. Consequently,

$$\sum_{i\in n} b(i)\pi(i) - \sum_{(i,j)\in A} \alpha_{ij}u_{ij} \leq \sum_{(i,j)\in A} c_{ij}x_{ij}. \tag{9.12}$$

Now notice that the left-hand side of (9.12) is the dual objective $w(\pi, \alpha)$ and the right-hand side is the primal objective, so we have established the lemma. ◆

The weak duality theorem implies that the objective function value of *any* dual feasible solution is a lower bound on the objective function value of *any* primal feasible solution. One consequence of this result is immediate: If some dual solution (π, α) and a primal solution x have the same objective function value, (π, α) must be an optimal solution of the dual problem and x must be an optimal solution of the primal problem (why?). Can we always find such solutions? The *strong duality theorem*, to be proved next, answers this question in the affirmative.

We first eliminate the dual variables α_{ij}'s from the dual formation (9.10) using some properties of the optimal solution. Defining the reduced cost, as before, as $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$, we can rewrite the constraint (9.10b) as

$$\alpha_{ij} \geq -c_{ij}^{\pi}. \tag{9.13}$$

The coefficient associated with the variable α_{ij} in the dual objective (9.10a) is $-u_{ij}$, and we wish to maximize the objective function value. Consequently, in any optimal solution we would assign the smallest possible value to α_{ij}. This observation, in view of (9.10c) and (9.13), implies that

$$\alpha_{ij} = \max\{0, -c_{ij}^{\pi}\}. \tag{9.14}$$

We have thus shown that if we know optimal values for the dual variables $\pi(i)$, we can compute the optimal values of the variables α_{ij} using (9.14). This construction permits us to eliminate the variables α_{ij} from the dual formulation. Substituting (9.14) in (9.10a) yields

$$\text{Maximize} \quad w(\pi) = \sum_{i \in N} b(i)\pi(i) - \sum_{(i,j) \in A} \max\{0, -c_{ij}^{\pi}\}u_{ij}. \tag{9.15}$$

The dual problem reduces to finding a vector π that optimizes (9.15). We are now in a position to prove the strong duality theorem. (Recall that our blanket assumption, Assumption 9.3, implies that the minimum cost flow problem always has a solution.)

Theorem 9.6 (Strong Duality Theorem). *For any choice of problem data, the minimum cost flow problem always has a solution x^* and the dual minimum cost flow problem has a solution π satisfying the property that $z(x^*) = w(\pi)$.*

Proof. We prove this theorem using the complementary slackness optimality conditions (9.8). Let x^* be an optimal solution of the minimum cost flow problem. Theorem 9.4 implies that x^* together with some vector π of node potentials satisfy the complementary slackness optimality conditions. We claim that this solution satisfies the condition

$$-c_{ij}^{\pi}x_{ij} = \max\{0, -c_{ij}^{\pi}\}u_{ij} \qquad \text{for every arc } (i, j) \in A. \tag{9.16}$$

To establish this result, consider the following three cases: (1) $c_{ij}^{\pi} > 0$, (2) $c_{ij}^{\pi} = 0$, and (3) $c_{ij}^{\pi} < 0$. The complementary slackness conditions (9.8) imply that in the first two cases, both the left-hand side and right-hand side of (9.16) are zero, and in the third case both sides equal $-c_{ij}^{\pi}u_{ij}$.

Next consider the dual objective (9.15). Substituting (9.16) in (9.15) yields

$$w(\pi) = \sum_{i \in N} b(i)\pi(i) + \sum_{(i,j) \in A} c_{ij}^{\pi}x_{ij}^* = \sum_{(i,j) \in A} c_{ij}x_{ij}^* = z(x^*).$$

The second last inequality follows from Property 2.4. This result is the conclusion of the theorem. ◆

The proof of this theorem shows that any optimal solution x^* of the minimum cost flow problem always has an associated dual solution π satisfying the condition $z(x^*) = w(\pi)$. Needless to say, the solution π is an optimal solution of the dual minimum cost flow problem since any larger value of the dual objective would contradict the weak duality theorem stated in Theorem 9.5.

In Theorem 9.6 we showed that the complementary slackness optimality conditions implies strong duality. We next prove the converse result: namely, that strong duality implies the complementary slackness optimality conditions.

Theorem 9.7. *If x is a feasible flow and π is an (arbitrary) vector satisfying the property that $z(x) = w(\pi)$, then the pair (x, π) satisfies the complementary slackness optimality conditions.*

Proof. Since $z(x) = w(\pi)$,

$$\sum_{(i,j)\in A} c_{ij}x_{ij} = \sum_{i\in N} b(i)\pi(i) - \sum_{(i,j)\in A} \max\{0, -c_{ij}^{\pi}\}u_{ij}. \tag{9.17}$$

Substituting the result of Property 2.4 in (9.17) shows that

$$\sum_{(i,j)\in A} \max\{0, -c_{ij}^{\pi}\}u_{ij} = \sum_{(i,j)\in A} -c_{ij}^{\pi}x_{ij}. \tag{9.18}$$

Now observe that both the sides have m terms, and each term on the left-hand side is nonnegative and its value is an upper bound on the corresponding term on the right-hand side (because $\max\{0, -c_{ij}^{\pi}\} \geq -c_{ij}^{\pi}$ and $u_{ij} \geq x_{ij}$). Therefore, the two sides can be equal only when

$$\max\{0, -c_{ij}^{\pi}\}u_{ij} = -c_{ij}^{\pi}x_{ij} \qquad \text{for every arc } (i, j) \in A. \tag{9.19}$$

Now we consider three cases.

(a) $c_{ij}^{\pi} > 0$. In this case, the left-hand side of (9.19) is zero, and the right-hand side can be zero only if $x_{ij} = 0$. This conclusion establishes (9.8a).

(b) $0 < x_{ij} < u_{ij}$. In this case, $c_{ij}^{\pi} = 0$; otherwise, the right-hand side of (9.19) is negative. This conclusion establishes (9.8b).

(c) $c_{ij}^{\pi} < 0$. In this case, the left-hand side of (9.19) is $-c_{ij}^{\pi}u_{ij}$ and therefore, $x_{ij} = u_{ij}$. This conclusion establishes (9.8c).

These results complete the proof of the theorem. ◆

The following result is an easy consequence of Theorems 9.6 and 9.7.

Property 9.8. *If x^* is an optimal solution of the minimum cost flow problem, and π is an optimal solution of the dual minimum cost flow problem, the pair (x^*, π) satisfies the complementary slackness optimality conditions* (9.8).

Proof. Theorem 9.6 implies that $z(x^*) = w(\pi)$ and Theorem 9.7 implies that the pair (x^*, π) satisfies (9.8). ◆

One important implication of the minimum cost flow duality is that it permits us to solve linear programs that have at most one $+1$ and at most one -1 in each row as minimum cost flow problems. Linear programs with this special structure arise in a variety of situations; Applications 19.10, 19.11, 19.18, and Exercises 9.9 and 19.18 provide a few examples.

Before examining the situation with at most one $+1$ and at most one -1 in each row, let us consider a linear program that has at most one $+1$ and at most one -1 in each column. We assume, without any loss of generality, that each constraint in the linear program is in equality form, because we can always bring the linear program into this form by introducing slack or surplus variables. (Observe that column corresponding the slack or surplus variables will also have one $+1$ or one -1.) If each column has exactly one $+1$ and exactly one -1, clearly the linear program is a minimum cost flow problem. Otherwise, we can augment this linear program by adding a redundant equality constraint which is the negative of the sum of all the

original constraints. (The new constraint corresponds to a new node that acts as a repository to deposit any excess supply or a source to fulfill any deficit demand from the other nodes.) The augmented linear program contains exactly one $+1$ and exactly one -1 in each column and the right-hand side values sum to zero. This model is clearly an instance of the minimum cost flow problem.

We now return to linear programs (in maximization form) that have at most one $+1$ and at most one -1 in each row. We allow a constraint in this linear program to be in any form: equality or inequality. The dual of this linear program contains at most one $+1$ and at most one -1 in each column, which we have already shown to be equivalent to a minimum cost flow problem. The variables in the dual problem will be nonnegative, nonpositive, or unrestricted, depending on whether they correspond to a less than or equal to, a greater than or equal to, or an equality constraint in the primal. A nonnegative variable x_{ij} defines a directed arc (i, j) in the resulting minimum cost flow formulation. To model any unrestricted variable x_{ij}, we replace it with two nonnegative variables, which is equivalent to introducing two arcs (i, j) and (j, i) of the same cost and capacity as this variable. The following theorem summarizes the preceding discussion.

Theorem 9.9. *Any linear program that contains* (a) *at most one* $+1$ *and at most one* -1 *in each column, or* (b) *at most one* $+1$ *and at most one* -1 *in each row, can be transformed into a minimum cost flow problem.* ◆

Minimum cost flow duality has several important implications. Since almost all algorithms for solving the primal problem also generate optimal node potentials $\pi(i)$ and the variables α_{ij}, solving the primal problem almost always solves both the primal and dual problems. Similarly, solving the dual problem typically solves the primal problem as well. Most algorithms for solving network flow problems explicitly or implicitly use properties of dual variables (since they are the node potentials that we have used at every turn) and of the dual linear program. In particular, the dual problem provides us with a certificate that if we can find a feasible dual solution that has the same objective function value as a given primal solution, we know from the strong duality theorem that the primal solution must be optimal, *without* making additional calculations and without considering other potentially optimal primal solutions. This certification procedure is a very powerful idea in network optimization, and in optimization in general. We have used it at many points in our previous developments and will see it many times again.

For network flow problems, the primal and dual problems are closely related via the basic shortest path and maximum flow problems that we have studied in previous chapters. In fact, these relationships help us to understand the fundamental importance of these two core problems to network flow theory and algorithms. We develop these relationships in the next section.

9.5 RELATING OPTIMAL FLOWS TO OPTIMAL NODE POTENTIALS

We next address the following questions: (1) Given an optimal flow, how might we obtain optimal node potentials? Conversely, (2) given optimal node potentials, how might we obtain an optimal flow? We show how to solve these problems by solving

either a shortest path problem or a maximum flow problem. These results point out an interesting relationship between the minimum cost flow problem and the maximum flow and shortest path problems.

Computing Optimal Node Potentials

We show that given an optimal flow x^*, we can obtain optimal node potentials by solving a shortest path problem (with possibly negative arc lengths). Let $G(x^*)$ denote the residual network with respect to the flow x^*. Clearly, $G(x^*)$ does not contain any negative cost cycle, for otherwise we would contradict the optimality of the solution x^*. Let $d(\cdot)$ denote the shortest path distances from node 1 to the rest of the nodes in the residual network if we use c_{ij} as arc lengths. The distances $d(\cdot)$ are well defined because the residual network does not contain a negative cycle. The shortest path optimality conditions (5.2) imply that

$$d(j) \leq d(i) + c_{ij} \qquad \text{for all } (i, j) \text{ in } G(x^*). \tag{9.20}$$

Let $\pi = -d$. Then we can restate (9.20) as

$$c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j) \geq 0 \qquad \text{for all } (i, j) \text{ in } G(x^*).$$

Theorem 9.3 shows that π constitutes an optimal set of node potentials.

Obtaining Optimal Flows

We now show that given a set of optimal node potentials π, we can obtain an optimal solution x^* by solving a maximum flow problem. First, we compute the reduced cost c_{ij}^{π} of every arc $(i, j) \in A$ and then we examine all arcs one by one. We classify each arc (i, j) in one of the following ways and use these categorizations of the arcs to define a maximum flow problem.

Case 1: $c_{ij}^{\pi} > 0$

The condition (9.8a) implies that x_{ij}^* must be zero. We enforce this constraint by setting $x_{ij}^* = 0$ and deleting arc (i, j) from the network.

Case 2: $c_{ij}^{\pi} < 0$

The condition (9.8c) implies that $x_{ij}^* = u_{ij}$. We enforce this constraint by setting $x_{ij}^* = u_{ij}$ and deleting arc (i, j) from the network. Since we sent u_{ij} units of flow on arc (i, j), we must decrease $b(i)$ by u_{ij} and increase $b(j)$ by u_{ij}.

Case 3: $c_{ij}^{\pi} = 0$

In this case we allow the flow on arc (i, j) to assume any value between 0 and u_{ij}.

Let $G' = (N, A')$ denote the resulting network and let b' denote the modified supplies/demands of the nodes. Now the problem reduces to finding a feasible flow in the network G' that meets the modified supplies/demands of the nodes. As noted in Section 6.2, we can find such a flow by solving a maximum flow problem defined as follows. We introduce a *source node* s, and a *sink node* t. For each node i with

$b'(i) > 0$, we add an arc (s, i) with capacity $b'(i)$ and for each node i with $b'(i) < 0$, we add an arc (i, t) with capacity $-b'(i)$. We now solve a maximum flow problem from node s to t in the transformed network obtaining a maximum flow x^*. The solution x^*_{ij} for all $(i, j) \in A$ is an optimal flow for the minimum cost flow problem in G.

9.6 CYCLE-CANCELING ALGORITHM AND THE INTEGRALITY PROPERTY

The negative cycle optimality conditions suggests one simple algorithmic approach for solving the minimum cost flow problem, which we call the *cycle-canceling algorithm*. This algorithm maintains a feasible solution and at every iteration attempts to improve its objective function value. The algorithm first establishes a feasible flow x in the network by solving a maximum flow problem (see Section 6.2). Then it iteratively finds negative cost-directed cycles in the residual network and augments flows on these cycles. The algorithm terminates when the residual network contains no negative cost-directed cycle. Theorem 9.1 implies that when the algorithm terminates, it has found a minimum cost flow. Figure 9.7 specifies this generic version of the cycle-canceling algorithm.

```
algorithm cycle-canceling;
begin
    establish a feasible flow x in the network;
    while G(x) contains a negative cycle do
    begin
        use some algorithm to identify a negative cycle W;
        δ : = min{r_ij : (i, j) ∈ W};
        augment δ units of flow in the cycle W and update G(x);
    end;
end;
```

Figure 9.7 Cycle canceling algorithm.

We use the example shown in Figure 9.8(a) to illustrate the cycle-canceling algorithm. (The reader might notice that our example does not satisfy Assumption 9.4; we violate this assumption so that the network is simpler to analyze.) Figure 9.8(a) depicts a feasible flow in the network and Figure 9.8(b) gives the corresponding residual network. Suppose that the algorithm first selects the cycle 4–2–3–4 whose cost is -1. The residual capacity of this cycle is 2. The algorithm augments 2 units of flow along this cycle. Figure 9.8(c) shows the modified residual network. In the next iteration, suppose that the algorithm selects the cycle 4–2–1–3–4 whose cost is -2. The algorithm sends 1 unit of flow along this cycle. Figure 9.8(d) depicts the updated residual network. Since this residual network contains no negative cycle, the algorithm terminates.

In Chapter 5 we discussed several algorithms for identifying a negative cycle if one exists. One algorithm for identifying a negative cycle is the FIFO label-correcting algorithm for the shortest path problem described in Section 5.4; this algorithm requires $O(nm)$ time. We describe other algorithms for detecting negative cycles in Sections 11.7 and 12.7.

A by-product of the cycle-canceling algorithm is the following important result.

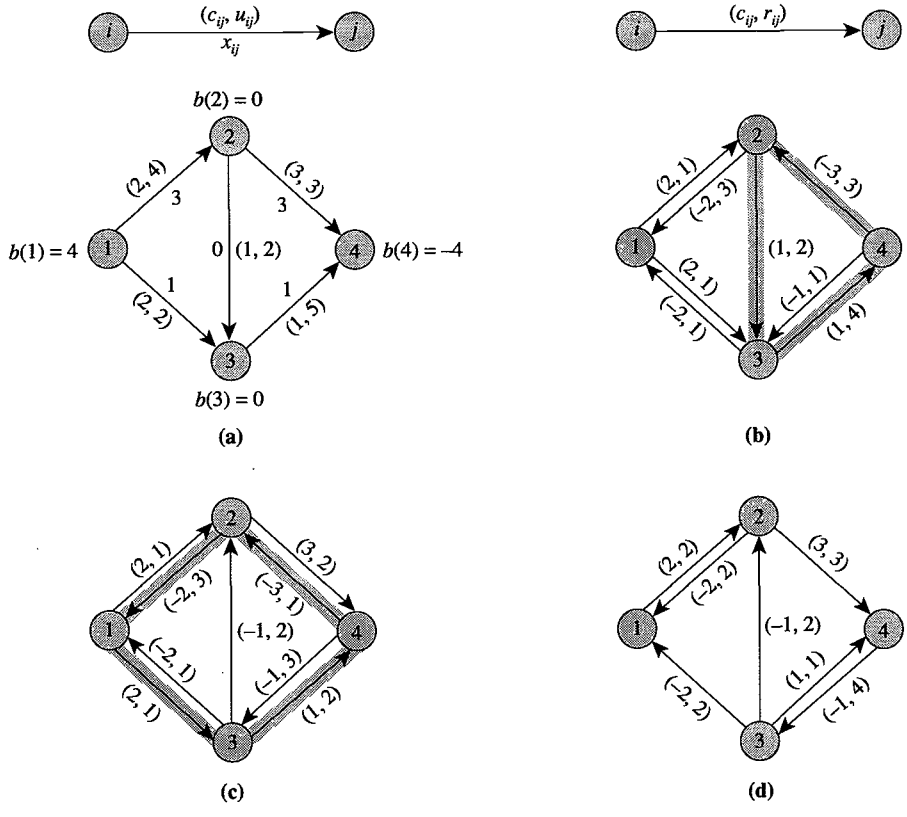


Figure 9.8 Illustrating the cycle canceling algorithm: (a) network example with a feasible flow x; (b) residual network $G(x)$; (c) residual network after augmenting 2 units along the cycle 4–2–3–4; (d) residual network after augmenting 1 unit along the cycle 4–2–1–3–4.

Theorem 9.10 (Integrality Property). *If all arc capacities and supplies/demands of nodes are integer, the minimum cost flow problem always has an integer minimum cost flow.*

Proof. We show this result by performing induction on the number of iterations. The algorithm first establishes a feasible flow in the network by solving a maximum flow problem. By Theorem 6.5 the problem has an integer feasible flow and we assume that the maximum flow algorithm finds an integer solution since all arc capacities in the network are integer and the initial residual capacities are also integer. The flow augmented by the cycle-canceling algorithm in any iteration equals the minimum residual capacity in the cycle canceled, which by the inductive hypothesis is integer. Therefore the modified residual capacities in the next iteration will again be integer. This conclusion implies the assertion of the theorem. ◆

Let us now consider the number of iterations that the algorithm performs. For the minimum cost flow problem, mCU is an upper bound on the initial flow cost

[since $c_{ij} \leq C$ and $x_{ij} \leq U$ for all $(i, j) \in A$] and $-mCU$ is a lower bound on the optimal flow cost [since $c_{ij} \geq -C$ and $x_{ij} \leq U$ for all $(i, j) \in A$]. Each iteration of the cycle-canceling algorithm changes the objective function value by an amount $(\sum_{(i,j)\in W} c_{ij})\delta$, which is strictly negative. Since we are assuming that all the data of the problem are integral, the algorithm terminates within $O(mCU)$ iterations and runs in $O(nm^2CU)$ time.

The generic version of the cycle-canceling algorithm does not specify the order for selecting negative cycles from the network. Different rules for selecting negative cycles produce different versions of the algorithm, each with different worse-case and theoretical behavior. The network simplex algorithm, which is widely considered to be one of the fastest algorithms for solving the minimum cost flow problem in practice, is a particular version of the cycle-canceling algorithm. The network simplex algorithm maintains information (a spanning tree solution and node potentials) that enables it to identify a negative cost cycle in $O(m)$ time. However, due to *degeneracy*, the algorithm cannot necessarily send a positive amount of flow along this cycle. We discuss these issues in Chapter 11, where we consider the network simplex algorithm in more detail. The most general implementation of the network simplex algorithm does not run in polynomial time. The following two versions of the cycle-canceling algorithm are, however, polynomial-time implementations.

Augmenting flow in a negative cycle with maximum improvement. Let x be any feasible flow and let x^* be an optimal flow. The improvement in the objective function value due to an augmentation along a cycle W is $-(\sum_{(i,j)\in W} c_{ij})$ $(\min\{r_{ij} : (i, j) \in W\})$. We observed in the proof of Theorem 3.7 in Section 3.5 that x^* equals x plus the flow on at most m augmenting cycles with respect to x, and improvements in cost due to flow augmentations on these augmenting cycles sum to $cx - cx^*$. Consequently, at least one of these augmenting cycles with respect to x must decrease the objective function value by at least $(cx - cx^*)/m$. Consequently, if the algorithm always augments flow along a cycle giving the maximum possible improvement, then Theorem 3.1 implies that the method would obtain an optimal flow within $O(m \log(mCU))$ iterations. Finding a maximum improvement cycle is difficult (i.e., it is a $\mathcal{NP}$-complete problem), but a modest variation of this approach yields a polynomial-time algorithm for the minimum cost flow problem. We provide a reference for this algorithm in the reference notes.

Augmenting flow along a negative cycle with minimum mean cost. We define the *mean cost* of a cycle as its cost divided by the number of arcs it contains. A *minimum mean cycle* is a cycle whose mean cost is as small as possible. It is possible to identify a minimum mean cycle in $O(nm)$ or $O(\sqrt{n}\, m \log(nC))$ time (see the reference notes of Chapter 5). Researchers have shown that if the cycle-canceling algorithm always augments flow along a minimum mean cycle, it performs $O(\min\{nm \log(nC), nm^2 \log n\})$ iterations. We describe this algorithm in Section 10.5.

9.7 SUCCESSIVE SHORTEST PATH ALGORITHM

The cycle-canceling algorithm maintains feasibility of the solution at every step and attempts to achieve optimality. In contrast, the successive shortest path algorithm maintains optimality of the solution (as defined in Theorem 9.3) at every step and strives to attain feasibility. It maintains a solution x that satisfies the nonnegativity and capacity constraints, but violates the mass balance constraints of the nodes. At each step, the algorithm selects a node s with excess supply (i.e., supply not yet sent to some demand node) and a node t with unfulfilled demand and sends flow from s to t along a shortest path in the residual network. The algorithm terminates when the current solution satisfies all the mass balance constraints.

To describe this algorithm as well as several later developments, we first introduce the concept of *pseudoflows*. A *pseudoflow* is a function $x: A \rightarrow R^+$ satisfying only the capacity and nonnegativity constraints; it need not satisfy the mass balance constraints. For any pseudoflow x, we define the *imbalance* of node i as

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

If $e(i) > 0$ for some node i, we refer to $e(i)$ as the *excess* of node i; if $e(i) < 0$, we call $-e(i)$ the node's *deficit*. We refer to a node i with $e(i) = 0$ as *balanced*. Let E and D denote the sets of excess and deficit nodes in the network. Notice that $\sum_{i\in N} e(i) = \sum_{i\in N} b(i) = 0$, and hence $\sum_{i\in E} e(i) = -\sum_{i\in D} e(i)$. Consequently, if the network contains an excess node, it must also contain a deficit node. The residual network corresponding to a pseudoflow is defined in the same way that we define the residual network for a flow.

Using the concept of pseudoflow and the reduced cost optimality conditions specified in Theorem 9.3, we next prove some results that we will use extensively in this and the following chapters.

Lemma 9.11. *Suppose that a pseudoflow (or a flow) x satisfies the reduced cost optimality conditions with respect to some node potentials π. Let the vector d represent the shortest path distances from some node s to all other nodes in the residual network $G(x)$ with c_{ij}^{π} as the length of an arc (i, j). Then the following properties are valid:*

(a) *The pseudoflow x also satisfies the reduced cost optimality conditions with respect to the node potentials $\pi' = \pi - d$.*
(b) *The reduced costs $c_{ij}^{\pi'}$ are zero for all arcs (i, j) in a shortest path from node s to every other node.*

Proof. Since x satisfies the reduced cost optimality conditions with respect to π, $c_{ij}^{\pi} \geq 0$ for every arc (i, j) in $G(x)$. Furthermore, since the vector d represents shortest path distances with c_{ij}^{π} as arc lengths, it satisfies the shortest path optimality conditions, that is,

$$d(j) \leq d(i) + c_{ij}^{\pi} \qquad \text{for all } (i, j) \text{ in } G(x). \tag{9.21}$$

Substituting $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ in (9.21), we obtain $d(j) \leq d(i) + c_{ij} - \pi(i) + \pi(j)$. Alternatively, $c_{ij} - (\pi(i) - d(i)) + (\pi(j) - d(j)) \geq 0$, or $c_{ij}^{\pi'} \geq 0$. This conclusion establishes part (a) of the lemma.

Consider next a shortest path from node s to some node l. For each arc (i, j) in this path, $d(j) = d(i) + c_{ij}^{\pi}$. Substituting $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ in this equation, we obtain $c_{ij}^{\pi'} = 0$. This conclusion establishes part (b) of the lemma. ◆

The following result is an immediate corollary of the preceding lemma.

Lemma 9.12. *Suppose that a pseudoflow (or a flow) x satisfies the reduced cost optimality conditions and we obtain x' from x by sending flow along a shortest path from node s to some other node k; then x' also satisfies the reduced cost optimality conditions.*

Proof. Define the potentials π and π' as in Lemma 9.11. The proof of Lemma 9.11 implies that for every arc (i, j) in the shortest path P from node s to the node k, $c_{ij}^{\pi'} = 0$. Augmenting flow on any such arc might add its reversal (j, i) to the residual network. But since $c_{ij}^{\pi'} = 0$ for each arc $(i, j) \in P$, $c_{ji}^{\pi'} = 0$ and the arc (j, i) also satisfies the reduced cost optimality conditions. These results establish the lemma. ◆

We are now in a position to describe the successive shortest path algorithm. The node potentials play a very important role in this algorithm. Besides using them to prove the correctness of the algorithm, we use them to maintain nonnegative arc lengths so that we can solve the shortest path problem more efficiently. Figure 9.9 gives a formal statement of the successive shortest path algorithm.

We illustrate the successive shortest path algorithm on the same numerical example we used to illustrate the cycle canceling algorithm. Figure 9.10(a) shows the initial residual network. Initially, $E = \{1\}$ and $D = \{4\}$. Therefore, in the first iteration, $s = 1$ and $t = 4$. The shortest path distances d (with respect to the reduced costs) are $d = (0, 2, 2, 3)$ and the shortest path from node 1 to node 4 is 1–3–4. Figure 9.10(b) shows the updated node potentials and reduced costs, and Figure 9.10(c) shows the solution after we have augmented $\min\{e(1), -e(4), r_{13}, r_{34}\} = \min\{4, 4, 2, 5\} = 2$ units of flow along the path 1–3–4. In the second iteration, $k =$

```
algorithm successive shortest path;
begin
    x := 0 and π := 0;
    e(i) := b(i) for all i ∈ N;
    initialize the sets E := {i : e(i) > 0} and D := {i : e(i) < 0};
    while E ≠ ∅ do
    begin
        select a node k ∈ E and a node l ∈ D;
        determine shortest path distances d(j) from node s to all
            other nodes in G(x) with respect to the reduced costs c_ij^π;
        let P denote a shortest path from node k to node l;
        update π := π − d;
        δ := min[e(k), − e(l), min{r_ij : (i, j) ∈ P}];
        augment δ units of flow along the path P;
        update x, G(x), E, D, and the reduced costs;
    end;
end;
```

Figure 9.9 Successive shortest path algorithm.

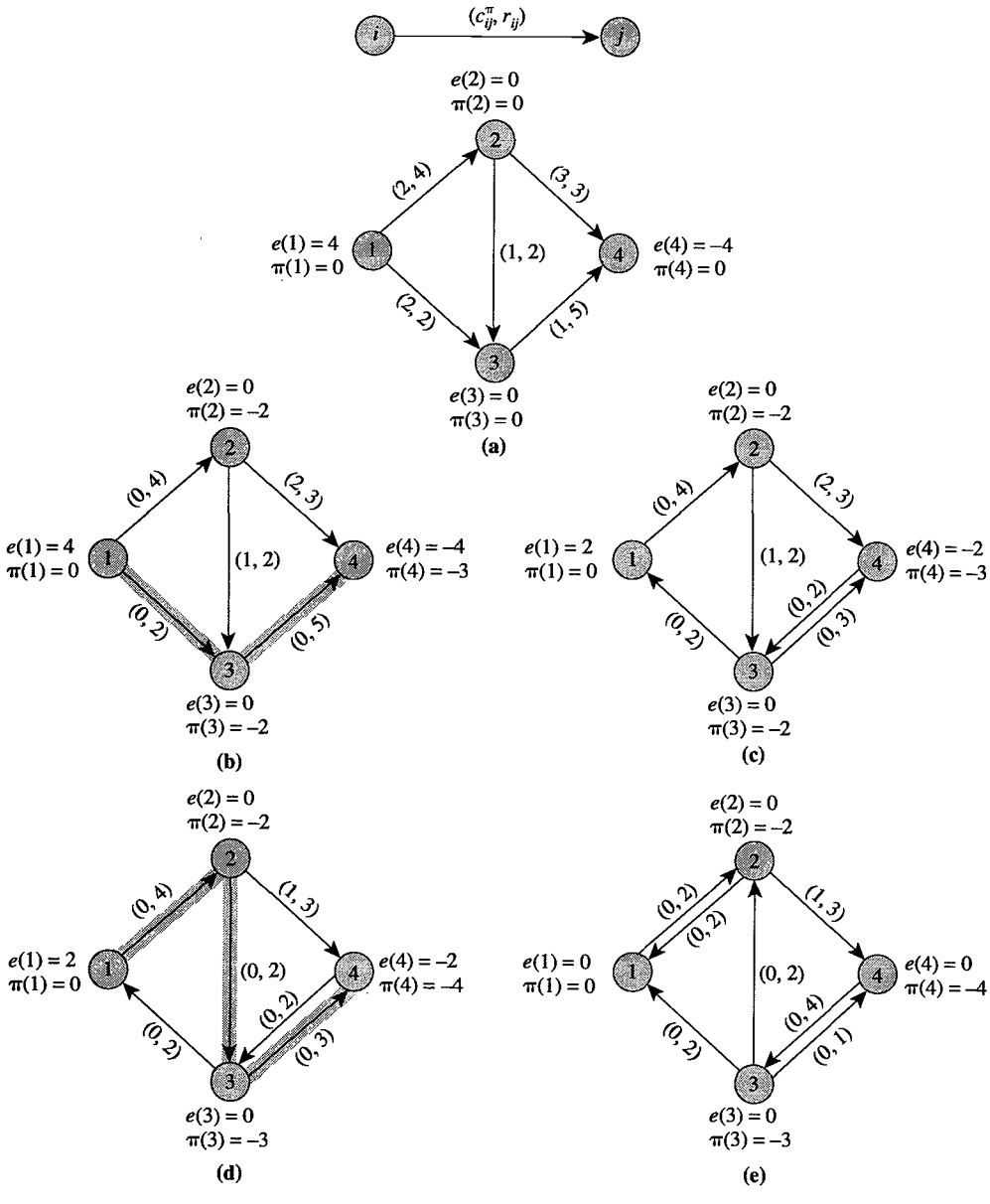


Figure 9.10 Illustrating the successive shortest path algorithm: (a) initial residual network for $x = 0$ and $\pi = 0$; (b) network after updating the potentials π; (c) network after augmenting 2 units along the path 1–3–4; (d) network after updating the potentials π; (e) network after augmenting 2 units along the path 1–2–3–4.

1, $l = 4$, $d = (0, 0, 1, 1)$ and the shortest path from node 1 to node 4 is 1–2–3–4. Figure 9.10(d) shows the updated node potentials and reduced costs, and Figure 9.10(e) shows the solution after we have augmented $\min\{e(1), -e(4), r_{12}, r_{23}, r_{34}\} = \min\{2, 2, 4, 2, 3\} = 2$ units of flow. At the end of this iteration, all imbalances become zero and the algorithm terminates.

We now justify the successive shortest path algorithm. To initialize the algorithm, we set $x = 0$, which is a feasible pseudoflow. For the zero pseudoflow x, $G(x) = G$. Note that this solution together with $\pi = 0$ satisfies the reduced cost optimality conditions because $c_{ij}^{\pi} = c_{ij} \geq 0$ for every arc (i, j) in the residual network $G(x)$ (recall Assumption 9.5, which states that all arc costs are nonnegative). Observe that as long as any node has a nonzero imbalance, both E and D must be nonempty since the total sum of excesses equals the total sum of deficits. Thus until all nodes are balanced, the algorithm always succeeds in identifying an excess node k and a deficit node l. Assumption 9.4 implies that the residual network contains a directed path from node k to every other node, including node l. Therefore, the shortest path distances $d(\cdot)$ are well defined. Each iteration of the algorithm solves a shortest path problem with nonnegative arc lengths and strictly decreases the excess of some node (and, also, the deficit of some other node). Consequently, if U is an upper bound on the largest supply of any node, the algorithm would terminate in at most nU iterations. If $S(n, m, C)$ denotes the time taken to solve a shortest path problem with nonnegative arc lengths, the overall complexity of this algorithm is $O(nUS(n, m, nC))$. [Note that we have used nC rather than C in this expression, since the costs in the residual network are bounded by nC.] We refer the reader to the reference notes of Chapter 4 for the best available value of $S(n, m, C)$.

The successive shortest path algorithm requires pseudopolynomial time to solve the minimum cost flow problem since it is polynomial in n, m and the largest supply U. This algorithm is, however, polynomial time for the assignment problem, a special case of the minimum cost flow problem, for which $U = 1$. In Chapter 10, using scaling techniques, we develop weakly and strongly polynomial-time versions of the successive shortest path algorithm. In Section 14.5 we generalize this approach even further, developing a polynomial-time algorithm for the convex cost flow problem.

We now suggest some practical improvements to the successive shortest path algorithm. As stated, this algorithm selects an excess node k, uses Dijkstra's algorithm to identify shortest paths from node k to all other nodes, and augments flow along a shortest path from node k to some deficit node l. In fact, it is not necessary to determine a shortest path from node k to *all* nodes; a shortest path from node k to *one deficit node* l is sufficient. Consequently, we could terminate Dijkstra's algorithm whenever it permanently labels the first deficit node l. At this point we might modify the node potentials in the following manner:

$$\pi(i) = \begin{cases} \pi(i) - d(i) & \text{if node } i \text{ is permanently labeled} \\ \pi(i) - d(l) & \text{if node } i \text{ is temporarily labeled.} \end{cases}$$

In Exercise 9.47 we ask the reader to show that with this choice of the modified node potentials, the reduced costs of all the arcs in the residual network remain nonnegative and the reduced costs of the arcs along the shortest path from node k to node l are zero. Observe that we can alternatively modify the node potentials in the following manner:

$$\pi(i) = \begin{cases} \pi(i) - d(i) + d(l) & \text{if node } i \text{ is permanently labeled} \\ \pi(i) & \text{if node } i \text{ is temporarily labeled.} \end{cases}$$

This scheme for updating node potentials is the same as the previous scheme except that we add $d(l)$ to all of the node potentials (which does not affect the reduced cost of any arc). An advantage of this scheme is that the algorithm spends no time updating the potentials of the temporarily labeled nodes.

9.8 PRIMAL–DUAL ALGORITHM

The primal–dual algorithm for the minimum cost flow problem is similar to the successive shortest path algorithm in the sense that it also maintains a pseudoflow that satisfies the reduced cost optimality conditions and gradually converts it into a flow by augmenting flows along shortest paths. In contrast, instead of sending flow along one shortest path at a time, it solves a maximum flow problem that sends flow along all shortest paths.

The primal–dual algorithm generally transforms the minimum cost flow problem into a problem with a single excess node and a single deficit node. We transform the problem into this form by introducing a *source* node s and a *sink* node t. For each node i with $b(i) > 0$, we add a zero cost arc (s, i) with capacity $b(i)$, and for each node i with $b(i) < 0$, we add a zero cost arc (i, t) with capacity $-b(i)$. Finally, we set $b(s) = \sum_{\{i \in N: b(i) > 0\}} b(i)$, $b(t) = -b(s)$, and $b(i) = 0$ for all $i \in N$. It is easy to see that a minimum cost flow in the transformed network gives a minimum cost flow in the original network. For simplicity of notation, we shall represent the transformed network as $G = (N, A)$, which is the same representation that we used for the original network.

The primal–dual algorithm solves a maximum flow problem on a subgraph of the residual network $G(x)$, called the *admissible network*, which we represent as $G^0(x)$. We define the admissible network $G^0(x)$ with respect to a pseudoflow x that satisfies the reduced cost optimality conditions for some node potentials π; the admissible network contains only those arcs in $G(x)$ with a zero reduced cost. The residual capacity of an arc in $G^0(x)$ is the same as that in $G(x)$. Observe that every directed path from node s to node t in $G^0(x)$ is a shortest path in $G(x)$ between the same pair of nodes (see Exercise 5.20). Figure 9.11 formally describes the primal–dual algorithm on the transformed network.

```
algorithm primal–dual;
begin
    x := 0 and π := 0;
    e(s) := b(s) and e(t) := b(t);
    while e(s) > 0 do
    begin
        determine shortest path distances d(·) from node s to all other nodes in G(x) with
            respect to the reduced costs c^π_ij;
        update π := π − d;
        define the admissible network G°(x);
        establish a maximum flow from node s to node t in G°(x);
        update e(s), e(t), and G(x);
    end;
end;
```

Figure 9.11 Primal–dual algorithm.

To illustrate the primal–dual algorithm, we consider the numerical example shown in Figure 9.12(a). Figure 9.12(b) shows the transformed network. The shortest path computation yields the vector $d = (0, 0, 0, 1, 2, 1)$ whose components are in

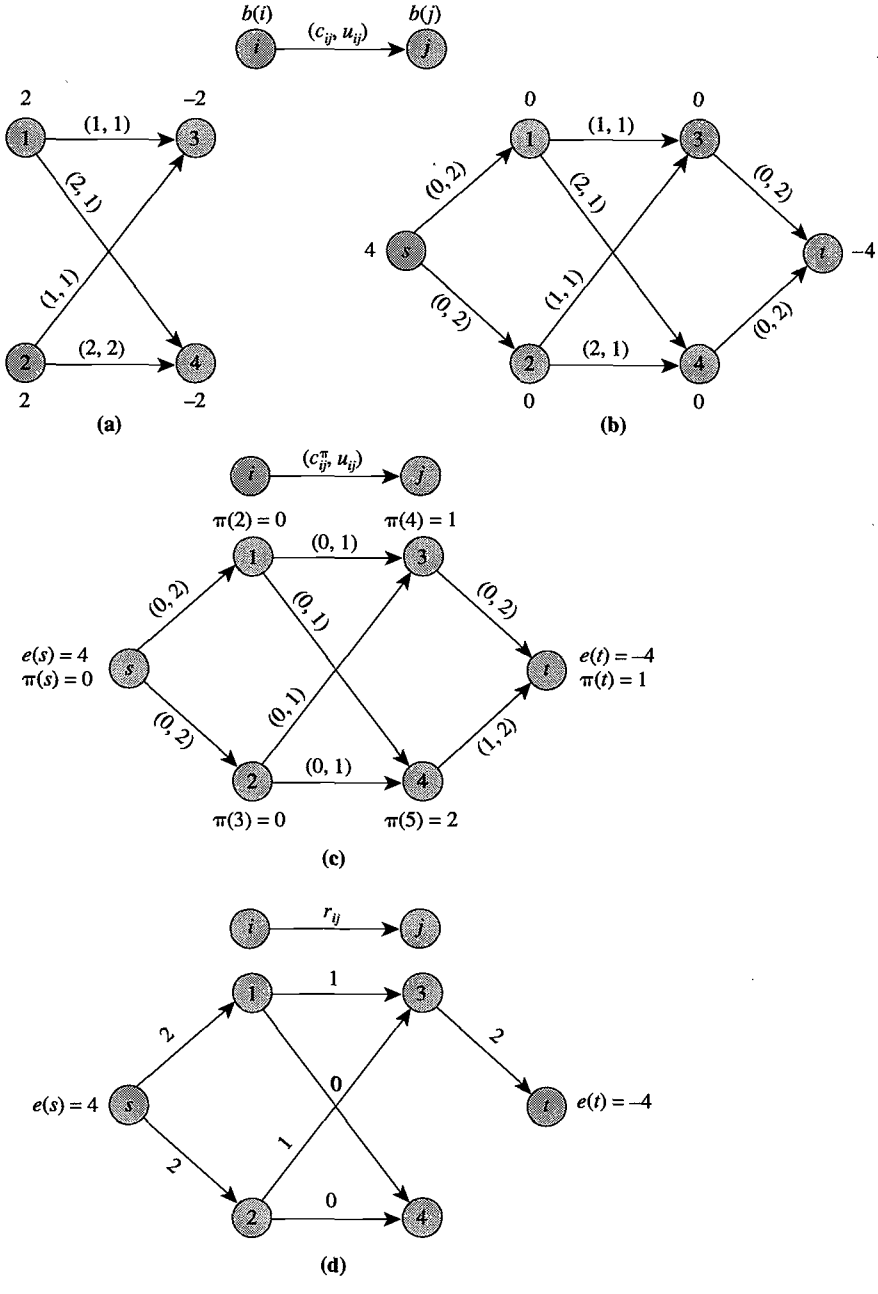


Figure 9.12 Illustrating the primal–dual algorithm: (a) example network; (b) transformed network; (c) residual network after updating the node potentials; (d) admissible network.

the order s, 1, 2, 3, 4, t. Figure 9.12(c) shows the modified node potentials and reduced costs and Figure 9.12(d) shows the admissible network at this stage in the computations. When we apply the maximum flow algorithm to the admissible network, it is able to send 2 units of flow from node s to node t. Observe that the admissible network contained two paths from node s to node t and the maximum flow computation saturates both the paths. The successive shortest path algorithm would have taken two iterations to send the 2 units of flow. As the reader can verify, the second iteration of the primal–dual algorithm also sends 2 units of flow from node s to node t, at which point it converts the pseudoflow into a flow and terminates.

The primal–dual algorithm guarantees that the excess of node s strictly decreases at each iteration, and also assures that the node potential of the sink strictly decreases from one iteration to the next. The second observation follows from the fact that once we have established a maximum flow in $G^{\circ}(x)$, the residual network $G(x)$ contains no directed path from node s to node t consisting entirely of arcs with zero reduced costs. Consequently, in the next iteration, when we solve the shortest path problem, $d(t) \geq 1$. These observations give a bound of $\min\{nU, nC\}$ on the number of iterations since initially $e(s) \leq nU$, and the value of no node potential can fall below $-nC$ (see Exercise 9.25). This bound on the number of iterations is better than that of the successive shortest path algorithm, but, of course, the algorithm incurs the additional expense of solving a maximum flow problem at every iteration. If $S(n, m, C)$ and $M(n, m, U)$ denote the solution times of shortest path and the maximum flow algorithms, the primal–dual algorithm has an overall complexity of $O(\min\{nU, nC\} \cdot \{S(n, m, nC) + M(n, m, U)\})$.

In concluding this discussion, we might comment on why this algorithm is known as the primal–dual algorithm. This name stems from linear programming duality theory. In the linear programming literature, the primal–dual algorithm always maintains a dual feasible solution π and a primal solution that might violate some supply/demand constraints (i.e., is primal infeasible), so that the pair satisfies the complementary slackness conditions. For a given dual feasible solution, the algorithm attempts to decrease the degree of primal infeasibility to the minimum possible level. [Recall that the algorithm solves a maximum flow problem to reduce $e(s)$ by the maximum amount.] When no further reduction in the primal infeasibility is possible, the algorithm modifies the dual solution (i.e., node potentials in the network flow context) and again tries to minimize primal infeasibility. This primal–dual approach is applicable to several combinatorial optimization problems and also to the general linear programming problem. Indeed, this primal–dual solution strategy is one of the most popular approaches for solving specially structured problems and has often yielded fairly efficient and intuitively appealing algorithms.

9.9 OUT-OF-KILTER ALGORITHM

The successive shortest path and primal–dual algorithms maintain a solution that satisfies the reduced cost optimality conditions and the flow bound constraints but violates the mass balance constraints. These algorithms iteratively modify arc flows and node potentials so that the flow at each step comes closer to satisfying the mass balance constraints. However, we could just as well have developed other solution strategies by violating other constraints at intermediate steps. The out-of-kilter al-

gorithm, which we discuss in this section, satisfies only the mass balance constraints, so intermediate solutions might violate both the optimality conditions and the flow bound restrictions. The algorithm iteratively modifies flows and potentials in a way that decreases the infeasibility of the solution (in a way to be specified) and, simultaneously, moves it closer to optimality. In essence, the out-of-kilter algorithm is similar to the successive shortest path and primal–dual algorithms because its fundamental step at every iteration is solving a shortest path problem and augmenting flow along a shortest path.

To describe the out-of-kilter algorithm, we refer to the complementary slackness optimality conditions stated in Theorem 9.4. For ease of reference, let us restate these conditions.

$$\text{If } x_{ij} = 0, \text{ then } c_{ij}^{\pi} \geq 0. \tag{9.22a}$$

$$\text{If } 0 < x_{ij} < u_{ij}, \text{ then } c_{ij}^{\pi} = 0. \tag{9.22b}$$

$$\text{If } x_{ij} = u_{ij}, \text{ then } c_{ij}^{\pi} \leq 0, \tag{9.22c}$$

The name *out-of-kilter algorithm* reflects the fact that arcs in the network either satisfy the complementary slackness optimality conditions (are *in-kilter*) or do not (are *out-of-kilter*). The so-called *kilter diagram* is a convenient way to represent these conditions. As shown in Figure 9.13, the kilter diagram of an arc (i, j) is the collection of all points (x_{ij}, c_{ij}^{π}) in the two-dimensional plane that satisfy the optimality conditions (9.22). The condition 9.22(a) implies that $c_{ij}^{\pi} \geq 0$ if $x_{ij} = 0$; therefore, the kilter diagram contains all points with zero x_{ij}-coordinates and nonnegative c_{ij}^{π}-coordinates. Similarly, the condition 9.22(b) yields the horizontal segment of the diagram, and condition 9.22(c) yields the other vertical segment of the diagram. Each arc has its own kilter diagram.

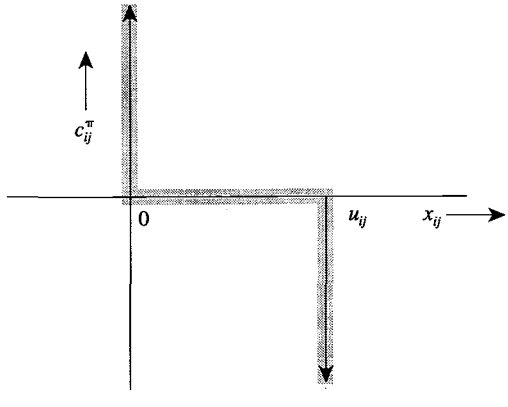


Figure 9.13 Kilter diagram for arc (i, j).

Notice that for every arc (i, j), the flow x_{ij} and reduced cost c_{ij}^{π} define a point (x_{ij}, c_{ij}^{π}) in the two-dimensional plane. If the point (x_{ij}, c_{ij}^{π}) lies on the thick lines in the kilter diagram, the arc is in-kilter; otherwise, it is out-of-kilter. For instance, the points B, D, and E in Figure 9.14 are in-kilter, whereas the points A and C are out-of-kilter. We define the *kilter number* k_{ij} of each arc (i, j) in A as the magnitude of the change in x_{ij} required to make the arc an in-kilter arc while keeping c_{ij}^{π} fixed.

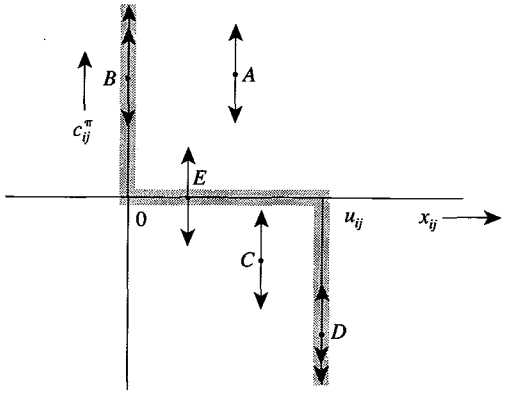


Figure 9.14 Examples of in-kilter and out-of-kilter arcs.

Therefore, in accordance with conditions (9.22a) and (9.22c), if $c_{ij}^{\pi} > 0$, then $k = | x_{ij} |$, and if $c_{ij}^{\pi} < 0$, then $k_{ij} = | u_{ij} - x_{ij} |$. If $c_{ij}^{\pi} = 0$ and $x_{ij} > u_{ij}$, then $k_{ij} = x_{ij} - u_{ij}$. If $c_{ij}^{\pi} = 0$ and $x_{ij} < 0$, then $k_{ij} = -x_{ij}$. The kilter number of any in-kilter arc is zero. The sum $K = \sum_{(i,j)\in A} k_{ij}$ of all kilter numbers provides us with a measure of how far the current solution is from optimality; the smaller the value of K, the closer the current solution is to being an optimal solution.

In describing the out-of-kilter algorithm, we begin by making a simplifying assumption that the algorithm starts with a feasible flow. At the end of this section we show how to extend the algorithm so that it applies to situations when the initial flow does not satisfy the arc flow bounds (we also consider situations with nonzero lower bounds on arc flows).

To describe the out-of-kilter algorithm, we will work on the residual network; in this setting, the algorithm iteratively decreases the kilter number of one or more arcs in the residual network. To do so, we must be able to define the kilter number of the arcs in the residual network $G(x)$. We set the kilter number k_{ij} of an arc (i, j) in the following manner:

$$k_{ij} = \begin{cases} 0 & \text{if } c_{ij}^{\pi} \geq 0. \\ r_{ij} & \text{if } c_{ij}^{\pi} < 0. \end{cases} \tag{9.23}$$

This definition of the kilter number of an arc in the residual network is consistent with our previous definition: It is the change in flow (or, equivalently, the residual capacity) required so that the arc satisfies its optimality condition [which, in the case of residual networks, is the reduced cost optimality condition (9.7)]. An arc (i, j) in the residual network with $c_{ij}^{\pi} \geq 0$ satisfies its optimality condition (9.7), but an arc (i, j) with $c_{ij}^{\pi} < 0$ does not. In the latter case, we must send r_{ij} units of flow on the arc (i, j) so that it drops out of the residual network and thus satisfies its optimality condition.

The out-of-kilter algorithm maintains a feasible flow x and a set of node potentials π. We could obtain a feasible flow by solving a maximum flow problem (as described in Section 6.2) and start with $\pi = 0$. Subsequently, the algorithm maintains all of the in-kilter arcs as in-kilter arcs and successively transforms the out-of-kilter arcs into in-kilter arcs. The algorithm terminates when all arcs in the residual network become in-kilter. Figure 9.15 gives a formal description of the out-of-kilter algorithm.

```
algorithm out-of-kilter;
begin
    π : = 0;
    establish a feasible flow x in the network;
    define the residual network G(x) and compute the kilter numbers of arcs;
    while the network contains an out-of-kilter arc do
    begin
        select an out-of-kilter arc (p, q) in G(x);
        define the length of each arc (i, j) in G(x) as max{0, c_ij^π};
        let d(·) denote the shortest path distances from node q to all other nodes in
            G(x) − {(q, p)} and let P denote a shortest path from node q to node p;
        update π'(i) : = π(i) − d(i) for all i ∈ N;
        if c_pq^π' < 0 then
        begin
            W : = P ∪ {(p, q)};
            δ : = min{r_ij : (i, j) ∈ W};
            augment δ units of flow along W;
            update x, G(x), and the reduced costs;
      end;
    end;
end;
```

Figure 9.15 Out-of-kilter algorithm.

We now discuss the correctness and complexity of the out-of-kilter algorithm. The correctness argument of the algorithm uses the fact that kilter numbers of arcs are nonincreasing. Two operations in the algorithm affect the kilter numbers of arcs: updating node potentials and augmenting flow along the cycle W. In the next two lemmas we show that these operations do not increase the kilter number of any arc.

Lemma 9.13. *Updating the node potentials does not increase the kilter number of any arc in the residual network.*

Proof. Let π and π' denote the node potentials in the out-of-kilter algorithm before and after the update. The definition of the kilter numbers from (9.23) implies that the kilter number of an arc (i, j) can increase only if $c_{ij}^{\pi} \geq 0$ and $c_{ij}^{\pi'} < 0$. We show that this cannot happen. Consider any arc (i, j) with $c_{ij}^{\pi} \geq 0$. We wish to show that $c_{ij}^{\pi'} \geq 0$. Since $c_{pq}^{\pi} < 0$, $(i, j) \neq (p, q)$. Since the distances $d(\cdot)$ represent the shortest path distances with $\max\{0, c_{ij}^{\pi}\}$ as the length of arc (i, j), the shortest path distances satisfy the following shortest path optimality condition (see Section 5.2):

$$d(j) \leq d(i) + \max\{0, c_{ij}^{\pi}\} = d(i) + c_{ij}^{\pi}.$$

The equality in this expression is valid because, by assumption, $c_{ij}^{\pi} \geq 0$. The preceding expression shows that

$$c_{ij}^{\pi} + d(i) - d(j) = c_{ij}^{\pi'} \geq 0,$$

so each arc in the residual network with a nonnegative reduced cost has a nonnegative reduced cost after the potentials update, which implies the conclusion of the lemma. ◆

Lemma 9.14. *Augmenting flow along the directed cycle* $W = P \cup \{(p, q)\}$ *does not increase the kilter number of any arc in the residual network and strictly decreases the kilter number of the arc* (p, q).

Proof. Notice that the flow augmentation can change the kilter number of only the arcs in $W = P \cup \{(p, q)\}$ and their reversals. Since P is a shortest path in the residual network with $\max\{0, c_{ij}^{\pi}\}$ as the length of arc (i, j),

$$d(j) = d(i) + \max\{0, c_{ij}^{\pi}\} \geq d(i) + c_{ij}^{\pi} \qquad \text{for each arc } (i, j) \in P,$$

which, using $\pi' = \pi - d$ and the definition $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$, implies that

$$c_{ij}^{\pi'} \leq 0 \qquad \text{for each arc } (i, j) \in P.$$

Since the reduced cost of each arc (i, j) in P with respect to π' is nonpositive, the condition (9.23) shows that sending additional flow does not increase the arc's kilter number, but might decrease it. The flow augmentation might add the reversals of arcs in P, but since $c_{ij}^{\pi'} \leq 0$, the reversal of this arc (j, i) has $c_{ji}^{\pi'} \geq 0$, and therefore arc (j, i) is an in-kilter arc.

Finally, we consider arc (p, q). Recall from the algorithm description in Figure 9.15 that we augment flow along the arc (p, q) only if it is an out-of-kilter arc (i.e., $c_{pq}^{\pi'} < 0$). Since augmenting flow along the arc (p, q) decreases its residual capacity, the augmentation decreases this arc's kilter number. Since $c_{qp}^{\pi'} > 0$, arc (q, p) remains an in-kilter arc. These conclusions complete the proof of the lemma. ◆

The preceding two lemmas allow us to obtain a pseudopolynomial bound on the running time of the out-of-kilter algorithm. Initially, the kilter number of an arc is at most U; therefore, the sum of the kilter numbers is at most mU. At each iteration, the algorithm selects an arc, say (p, q), with a positive kilter number and either makes it an in-kilter arc during the potential update step or decreases its kilter number by the subsequent flow augmentation. Therefore, the sum of kilter numbers decreases by at least 1 unit at every iteration. Consequently, the algorithm terminates within $O(mU)$ iterations. The dominant computation within each iteration is solving a shortest path problem. Therefore, if $S(n, m, C)$ is the time required to solve a shortest path problem with nonnegative arc lengths, the out-of-kilter algorithm runs in $O(mU\, S(n, m, nC))$ time.

How might we modify the algorithm to handle situations when the arc flows do not necessarily satisfy their flow bounds? In examining this case we consider the more general problem setting by allowing the arcs to have nonzero lower bounds. Let l_{ij} denote the lower bound on the flow on arc $(i, j) \in A$. In this case, the complementary slackness optimality conditions become:

$$\text{If } x_{ij} = l_{ij}, \text{ then } c_{ij}^{\pi} \geq 0. \tag{9.24a}$$

$$\text{If } l_{ij} < x_{ij} < u_{ij}, \text{ then } c_{ij}^{\pi} = 0. \tag{9.24b}$$

$$\text{If } x_{ij} = u_{ij}, \text{ then } c_{ij}^{\pi} \leq 0. \tag{9.24c}$$

The thick lines in Figure 9.16 define the kilter diagram for this case. Consider arc (i, j). If the point (x_{ij}, c_{ij}^{π}) lies on the thick line in Figure 9.16, the arc is an in-kilter arc; otherwise it is an out-of-kilter arc. As earlier, we define the kilter number

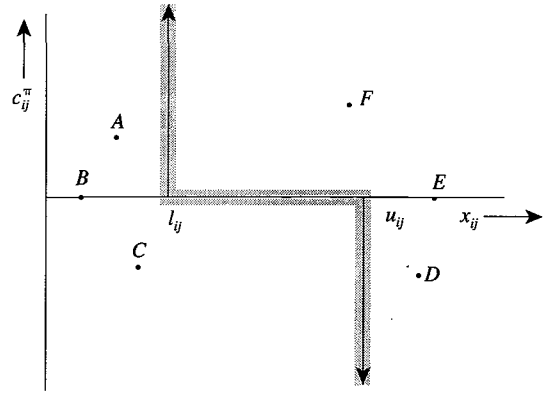


Figure 9.16 Kilter diagram for an arc (i, j) with a nonzero lower bound.

of an arc (i, j) in A as the magnitude of the change in x_{ij} required to make the arc an in-kilter arc while keeping c_{ij}^{π} fixed. Since arcs might violate their flow bounds, six types of out-of-kilter arcs are possible, which we depict by points A, B, C, D, E, and F in Figure 9.16. For example, the kilter numbers of arcs with coordinates depicted by the points A and D are $(l_{ij} - x_{ij})$ and $(x_{ij} - u_{ij})$, respectively.

To describe the algorithm for handling these situations, we need to determine how to form the residual network $G(x)$ for a flow x violating its lower and upper bounds. We consider each arc (i, j) in A one by one and add arcs to the residual network $G(x)$ in the following manner:

1. $l_{ij} \leq x_{ij} \leq u_{ij}$. If $x_{ij} < u_{ij}$, we add the arc (i, j) with a residual capacity $u_{ij} - x_{ij}$ and with a cost c_{ij}. If $x_{ij} > l_{ij}$, we add the arc (j, i) with a residual capacity $x_{ij} - l_{ij}$ and with a cost $-c_{ij}$. We call these arcs *feasible arcs*.
2. $x_{ij} < l_{ij}$. In this case we add the arc (i, j) with a residual capacity $(l_{ij} - x_{ij})$ and with a cost c_{ij}. We refer to this arc as a *lower-infeasible arc*.
3. $x_{ij} > u_{ij}$. In this case we add the arc (j, i) with a residual capacity $(x_{ij} - u_{ij})$ and with a cost $-c_{ij}$. We refer to this arc as an *upper-infeasible arc*.

We next define the kilter numbers of arcs in the residual network. For feasible arcs in the residual network, we define their kilter numbers using (9.23). We define the kilter number k_{ij} of a lower-infeasible or an upper-infeasible arc (i, j) as the change in its residual capacity required to restore its feasibility as well as its optimality. For instance, for a lower-infeasible arc (i, j) (1) if $c_{ij}^{\pi} \geq 0$, then $k_{ij} = (l_{ij} - x_{ij})$; and (2) if $c_{ij}^{\pi} < 0$, then $k_{ij} = (u_{ij} - x_{ij})$. Note that

1. Lower-infeasible and upper-infeasible arcs have positive kilter numbers.
2. Sending additional flow on lower-infeasible and upper-infeasible arcs in the residual network decreases their kilter numbers.

The out-of-kilter algorithm for this case is same as that for the earlier case. The algorithmic description given in Figure 9.15 applies to this case as well except that at the beginning of the algorithm we need not establish a feasible flow in the network. We can initiate the algorithm with $x = 0$ as the starting flow. We leave

the justification of the out-of-kilter algorithm for this case as an exercise to the reader (see Exercise 9.26).

9.10 RELAXATION ALGORITHM

All the minimum cost flow algorithms we have discussed so far—the cycle-canceling algorithm, the successive shortest path algorithm, the primal–dual algorithm, and the out-of-kilter algorithm—are classical in the sense that researchers developed them in the 1950s and 1960s as network flow area was emerging as an independent field of scientific investigation. These algorithms have several common features: (1) they repeatedly apply shortest path algorithms, (2) they run in pseudopolynomial time, and (3) their empirical running times have proven to be inferior to those of the network simplex algorithm tailored for the minimum cost flow problem (we discuss this algorithm in Chapter 11). The relaxation algorithm we examine in this section is a more recent vintage minimum cost flow algorithm; it is competitive or better than the network simplex algorithm for some classes of networks. Interestingly, the relaxation algorithm is also a variation of the successive shortest path algorithm. Even though the algorithm has proven to be efficient in practice for many classes of problems, its worst-case running time is much poorer than that of every minimum cost flow algorithm discussed in this chapter.

The relaxation algorithm uses ideas from *Lagrangian relaxation*, a well-known technique used for solving integer programming problems. We discuss the Lagrangian relaxation technique in more detail in Chapter 16. In the Lagrangian relaxation technique, we identify a set of constraints to be relaxed, multiply each such constraint by a scalar, and subtract the product from the objective function. The relaxation algorithm relaxes the mass balance constraints of the nodes, multiplying the mass balance constraint for node i by an (unrestricted) variable $\pi(i)$ (called, as usual, a node potential) and subtracts the resulting product from the objective function. These operations yield the following relaxed problem:

$$w(\pi) = \underset{x}{\text{minimize}} \left[\sum_{(i,j)\in A} c_{ij}x_{ij} + \sum_{i\in N} \pi(i) \left\{ - \sum_{\{j:(i,j)\in A\}} x_{ij} + \sum_{\{j:(j,i)\in A\}} x_{ji} + b(i) \right\} \right] \tag{9.25a}$$

subject to

$$0 \le x_{ij} \le u_{ij} \qquad \text{for all } (i, j) \in A. \tag{9.25b}$$

For a specific value of the vector π of node potentials, we refer to the relaxed problem as $LR(\pi)$ and denote its objective function value by $w(\pi)$. Note that the optimal solution of LR(π) is a pseudoflow for the minimum cost flow problem since it might violate the mass balance constraints. We can restate the objective function of LR(π) in the following equivalent way:

$$w(\pi) = \underset{x}{\text{minimize}} \left[\sum_{(i,j)\in A} c_{ij}x_{ij} + \sum_{i\in N} \pi(i)e(i) \right]. \tag{9.26}$$

In this expression, as in our earlier discussion, $e(i)$ denotes the imbalance of node i. Let us restate the objective function (9.25a) of the relaxed problem in another way. Notice that in the second term of (9.25a), each flow variable x_{ij} appears twice: once with a coefficient of $-\pi(i)$ and the second time with a coefficient of $\pi(j)$. Therefore, we can write (9.25a) as follows:

$$w(\pi) = \underset{x}{\text{minimize}} \left[\sum_{(i,j)\in A} (c_{ij} - \pi(i) + \pi(j))x_{ij} + \sum_{i\in N} \pi(i)b(i) \right],$$

or, equivalently,

$$w(\pi) = \underset{x}{\text{minimize}} \left[\sum_{(i,j)\in A} c_{ij}^{\pi} x_{ij} + \sum_{i\in N} \pi(i)b(i) \right]. \tag{9.27}$$

In the subsequent discussion, we refer to the objective function of LR(π) as (9.26) or (9.27), whichever is more convenient. For a given vector π of node potentials, it is very easy to obtain an optimal solution x of LR(π): In light of the formulation (9.27) of the objective function, (1) if $c_{ij}^{\pi} > 0$, we set $x_{ij} = 0$; (2) if $c_{ij}^{\pi} < 0$, we set $x_{ij} = u_{ij}$; and (3) if $c_{ij}^{\pi} = 0$, we can set x_{ij} to any value between 0 and u_{ij}. The resulting solution is a pseudoflow for the minimum cost flow problem and satisfies the reduced cost optimality conditions. We have therefore established the following result.

Property 9.15. *If a pseudoflow x of the minimum cost flow problem satisfies the reduced cost optimality conditions for some π, then x is an optimal solution of* LR(π).

Let z^* denote the optimal objective function value of the minimum cost flow problem. As shown by the next lemma, the value z^* is intimately related to the optimal objective value $w(\pi)$ of the relaxed problem LR(π).

Lemma 9.16

(a) *For any node potentials π, $w(\pi) \leq z^*$.*

(b) *For some choice of node potentials π^*, $w(\pi^*) = z^*$.*

Proof. Let x^* be an optimal solution of the minimum cost flow problem with objective function value z^*. Clearly, for any vector π of node potentials, x^* is a feasible solution of LR(π) and its objective function value in LR(π) is also z^*. Therefore, the minimum objective function value of LR(π) will be less than or equal to z^*. We have thus established the first part of the lemma.

To prove the second part, let π^* be a vector of node potentials that together with x^* satisfies the complementary slackness optimality conditions (9.8). Property 9.15 implies that x^* is an optimal solution of LR(π^*) and $w(\pi^*) = cx^* = z^*$. This conclusion completes the proof of the lemma. ◆

Notice the similarity between this result and the weak duality theorem (i.e., Theorem 9.5) for the minimum cost flow problem that we have stated earlier in this chapter. The similarity is more than incidental, since we can view the Lagrangian relaxation solution strategy as a dual linear programming approach that combines

some key features of both the primal and dual linear programs. Moreover, we can view the dual linear program itself as being generated by applying Lagrangian relaxation.

The relaxation algorithm always maintains a vector of node potentials π and a pseudoflow x that is an optimal solution of LR(π). In other words, the pair (x, π) satisfies the reduced cost optimality conditions. The algorithm repeatedly performs one of the following two operations:

1. Keeping π unchanged, it modifies x to x' so that x' is also an optimal solution of LR(π) and the excess of at least one node decreases.
2. It modifies π to π' and x to x' so that x' is an optimal solution of LR(π') and $w(\pi') > w(\pi)$.

If the algorithm can perform either of the two operations, it gives priority to the second operation. Consequently, the primary objective in the relaxation algorithm is to increase $w(\pi)$ and the secondary objective is to reduce the infeasibility of the pseudoflow x while keeping $w(\pi)$ unchanged. We point out that the excesses at the nodes might increase when the algorithm performs the second operation. As we show at the end of this section, these two operations are sufficient to guarantee finite convergence of the algorithm. For a fixed value of $w(\pi)$, the algorithm consistently reduces the excesses of the nodes by at least one unit, and from Lemma 9.16 the number of increases in $w(\pi)$, each of which is at least 1 unit, is finite.

We now describe the relaxation algorithm in more detail. The algorithm performs major iterations and, within a major iteration, it performs several minor iterations. Within a major iteration, the algorithm selects an excess node s and grows a tree rooted at node s so that every tree node has a nonnegative imbalance and every tree arc has zero reduced cost. Each minor iteration adds an additional node to the tree. A major iteration ends when the algorithm performs either an augmentation or increases $w(\pi)$.

Let S denote the set of nodes spanned by the tree at some stage and let $\bar{S} = N - S$. The set S defines a cut which we denote by $[S, \bar{S}]$. As in earlier chapters, we let $(S, \bar{S})$ denote the set of forward arcs in the cut and $(\bar{S}, S)$ the set of backward arcs [all in $G(x)$]. The algorithm maintains two variables $e(S)$ and $r(\pi, S)$, defined as follows:

$$e(S) = \sum_{i \in S} e(i),$$

$$r(\pi, S) = \sum_{(i,j) \in (S, \bar{S}) \text{ and } c_{ij}^{\pi} = 0} r_{ij}.$$

Given the set S, the algorithm first checks the condition $e(S) > r(\pi, S)$. If the current solution satisfies this condition, the algorithm can increase $w(\pi)$ in the following manner. [We illustrate this method using the example shown in Figure 9.17(a).] The algorithm first increases the flow on zero reduced cost arcs in $(S, \bar{S})$ so that they become saturated (i.e., drop out of the residual network). The flow change does not alter the value of $w(\pi)$ because the change takes place on arcs with zero reduced costs. However, the flow change decreases the total imbalance of the

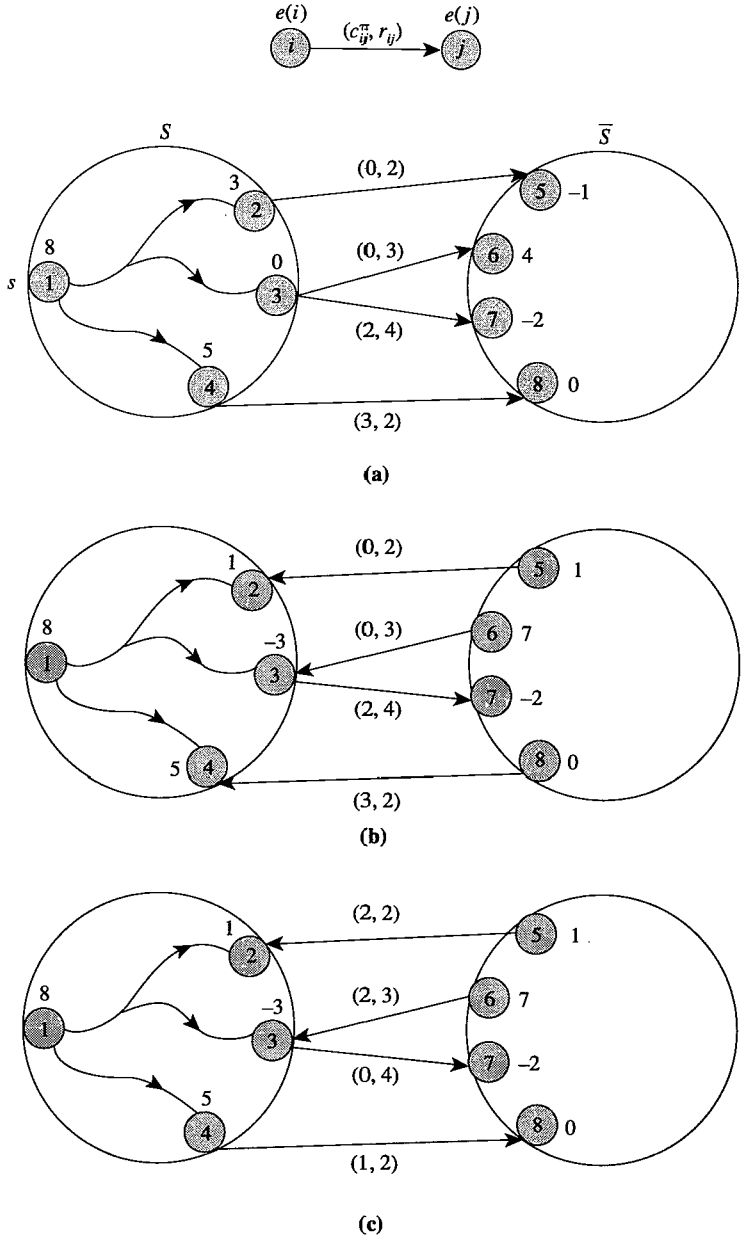


Figure 9.17 Illustrating the relaxation algorithm: (a) solution at some stage; (b) solution after modifying the flow; (c) solution after modifying the potentials.

nodes by the amount $r(\pi, S)$; but since $e(S) > r(\pi, S)$, the remaining imbalance $e(S) - r(\pi, S)$ is still positive [see Figure 9.17(b)].

At this point all the arcs in $(S, \bar{S})$ have (strictly) positive reduced cost. The algorithm next computes the minimum reduced cost of an arc in $(S, \bar{S})$, say α, and increases the potential of every node $i \in S$ by $\alpha > 0$ units [see Figure 9.17(c)]. The

formulation (9.26) of the Lagrangian relaxation objective function implies that this updating of the node potentials does not change its first term but increases the second term by $(e(S) - r(\pi, S))\alpha$ units. Therefore, this operation increases $w(\pi)$ by $(e(S) - r(\pi, S))\alpha$ units, which is strictly positive. Increasing the potentials of nodes in S by α decreases the reduced costs of all the arcs in $(S, \bar{S})$ by α units, increases the reduced costs of all arcs in $(\bar{S}, S)$ by α units, and does not change the remaining reduced costs. Although increasing the reduced costs does not change the reduced cost optimality conditions, decreasing the reduced costs might. Notice, however, that before we change the node potentials, $c_{ij}^{\pi} \geq \alpha$ for all $(i, j) \in (S, \bar{S})$; therefore, after the change, $c_{ij}^{\pi'} \geq 0$, so the algorithm preserves the optimality conditions. This completes one major iteration.

We next study situations in which $e(S) \leq r(\pi, S)$. Since $r(\pi, S) \geq e(S) > 0$, at least one arc $(i, j) \in (S, \bar{S})$ must have a zero reduced cost. If $e(j) \geq 0$, the algorithm adds node j to S, completes one minor iteration, and repeats this process. If $e(j) < 0$, the algorithm augments the maximum possible flow along the tree path from node s to node j. Notice that since we augment flow along zero residual cost arcs, we do not change the objective function value of LR(π). The augmentation reduces the total excess of the nodes and completes one major iteration of the algorithm.

Figures 9.18 and 9.19 give a formal description of the relaxation algorithm.

It is easy to see that the algorithm terminates with a minimum cost flow. The algorithm terminates when all of the node imbalances have become zero (i.e., the solution is a flow). Because the algorithm maintains the reduced cost optimality conditions at every iteration, the terminal solution is a minimum cost flow.

We now prove that for problems with integral data, the algorithm terminates in a finite number of iterations. Since each minor iteration adds a node to the set S, within n minor iterations the algorithm either executes adjust-flow or executes adjust-potentials. Each call of the procedure adjust-flow decreases the excess of at least one node by at least 1 unit; therefore, the algorithm can perform a finite number of executions of the adjust-flow procedure within two consecutive calls of the adjust-potential procedure. To bound the executions of the adjust-potential procedure, we notice that (1) initially, $w(\pi) = 0$; (2) each call of this procedure strictly increases

```
algorithm relaxation;
begin
    x := 0 and π := 0;
    while the network contains a node s with e(s) > 0 do
    begin
        S := {s};
        if e(S) > r(π, S) then adjust-potential;
        repeat
            select an arc (i, j) ∈ (S, S̄) in the residual network with c_ij^π = 0;
            if e(j) ≥ 0 then set pred(j) := i and add node j to S;
        until e(j) < 0 or e(S) > r(π, S);
        if e(S) > r(π, S) then adjust-potential
        else adjust-flow;
    end;
end;
```

Figure 9.18 Relaxation algorithm.

```
procedure adjust-potential;
begin
    for every arc (i, j) ∈ (S, S̄) with c^π_ij = 0 do send r_ij units of flow on the arc (i, j);
    compute α := min{c^π_ij : (i, j) ∈ (S, S̄) and r_ij > 0};
    for every node i ∈ S do π(i) := π(i) + α;
end;
```

(a)

```
procedure adjust-flow;
begin
    trace the predecessor indices to identify the directed path P from node s to node j;
    δ := min[e(s), − e(j), min{r_ij : (i, j) ∈ P}];
    augment δ units of flow along P, update imbalances and residual capacities;
end;
```

(b)

Figure 9.19 Procedures of the relaxation algorithm.

$w(\pi)$ by at least 1 unit; and (3) the maximum possible value of $w(\pi)$ is mCU. The preceding arguments establish that the algorithm performs finite number of iterations. In Exercise 9.27 we ask the reader to obtain a worst-case bound on the total number of iterations; this time bound is much worse than those of the other minimum cost flow algorithms discussed in earlier sections.

Notice that the relaxation algorithm is a type of shortest augmenting path algorithm; indeed, it bears some resemblance to the successive shortest path algorithm that we considered in Section 9.7. Since the reduced cost of every arc in the residual network is nonnegative, and since every arc in the tree connecting the nodes in S has a zero reduced cost, the path P that we find in the adjust-flow procedure of the relaxation algorithm is a shortest path in the residual network. Therefore, the sequence of flow adjustments that the algorithm makes is a set of flow augmentations along shortest augmenting paths. The relaxation algorithm differs from the successive shortest augmenting path algorithm, however, because it uses "intermediate" information to make changes to the node potentials as it fans out and constructs the tree containing the nodes S. This use of intermediate information might explain why the relaxation algorithm has performed much better empirically than the successive shortest path algorithm.

9.11 SENSITIVITY ANALYSIS

The purpose of sensitivity analysis is to determine changes in the optimal solution of a minimum cost flow problem resulting from changes in the data (supply/demand vector or the capacity or cost of any arc). There are two different ways of performing sensitivity analysis: (1) using combinatorial methods, and (2) using simplex-based methods from linear programming. Each method has its advantages. For example, although combinatorial methods obtain better worst-case time bounds for performing sensitivity analysis, simplex-based methods might be more efficient in practice. In this section we describe sensitivity analysis using combinatorial methods; in Section

11.10 we consider a simplex-based approach. For simplicity, we limit our discussion to a unit change of only a particular type. In a sense, however, this discussion is quite general: It is possible to reduce more complex changes to a sequence of the simple changes we consider. We show that sensitivity analysis for the minimum cost flow problem essentially reduces to applying shortest path or maximum flow algorithms.

Let x^* denote an optimal solution of a minimum cost flow problem. Let π be the corresponding node potentials and $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ denote the reduced costs. Further, let $d(k, l)$ denote the shortest distance from node k to node l in the residual network with respect to the original arc lengths c_{ij}. Since for any directed path P from node k to node l, $\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l)$, $d(k, l)$ equals the shortest distance from node k to node l with respect to the arc lengths c_{ij}^{π} plus $[\pi(k) - \pi(l)]$. At optimality, the reduced costs c_{ij}^{π} of all arcs in the residual network are nonnegative. Therefore, we can compute $d(k, l)$ for all pairs of nodes k and l by solving n single-source shortest path problems with nonnegative arc lengths.

Supply/Demand Sensitivity Analysis

We first study changes in the supply/demand vector. Suppose that the supply/demand of a node k becomes $b(k) + 1$ and the supply/demand of another node l becomes $b(l) - 1$. [Recall from Section 9.1 that feasibility of the minimum cost flow problem dictates that $\sum_{i\in N} b(i) = 0$; therefore, we must change the supply/demand values of two nodes by equal magnitudes, and must increase one value and decrease the other.] The vector x^* is a pseudoflow for the modified problem; moreover, this vector satisfies the reduced cost optimality conditions. Augmenting 1 unit of flow from node k to node l along the shortest path in the residual network $G(x^*)$ converts this pseudoflow into a flow. This augmentation changes the objective function value by $d(k, l)$ units. Lemma 9.12 implies that this flow is optimal for the modified minimum cost flow problem. We point out that the residual network $G(x^*)$ might not contain any directed path from node k to node l, in which case the modified minimum cost flow problem is infeasible.

Arc Capacity Sensitivity Analysis

We next consider a change in an arc capacity. Suppose that the capacity of an arc (p, q) increases by 1 unit. The flow x^* is feasible for the modified problem. In addition, if $c_{pq}^{\pi} \geq 0$, it satisfies the reduced cost optimality conditions; therefore, it is an optimal flow for the modified problem. If $c_{pq}^{\pi} < 0$, the optimality conditions dictate that the flow on the arc must equal its capacity. We satisfy this requirement by increasing the flow on the arc (p, q) by 1 unit, which produces a pseudoflow with an excess of 1 unit at node q and a deficit of 1 unit at node p. We convert the pseudoflow into a flow by augmenting 1 unit of flow from node q to node p along the shortest path in the residual network $G(x^*)$, which changes the objective function value by an amount $c_{pq} + d(q, p)$. This flow is optimal from our observations concerning supply/demand sensitivity analysis.

When the capacity of the arc (p, q) decreases by 1 unit and the flow on the arc is strictly less than its capacity, x^* remains feasible, and therefore optimal, for the modified problem. However, if the flow on the arc is at its capacity, we decrease

the flow by 1 unit and augment 1 unit of flow from node p to node q along the shortest path in the residual network. This augmentation changes the objective function value by an amount $-c_{pq} + d(p, q)$. Observed that the residual network $G(x^*)$ might not contain any directed path from node p to node q, indicating the infeasibility of the modified problem.

Cost Sensitivity Analysis

Finally, we discuss changes in arc costs, which we assume are integral. We discuss the case when the cost of an arc (p, q) increases by 1 unit; the case when the cost of an arc decreases is left as an exercise to the reader (see Exercise 9.50). This change increases the reduced cost of arc (p, q) by 1 unit as well. If $c^{\pi}_{pq} < 0$ before the change, then after the change, the modified reduced cost is nonpositive. Similarly, if $c^{\pi}_{pq} > 0$ before the change, the modified reduced cost is nonnegative after the change. In both cases we preserve the optimality conditions. However, if $c^{\pi}_{pq} = 0$ before the change and $x_{pq} > 0$, then after the change the modified reduced cost is positive and the solution violates the reduced-cost optimality conditions. To restore the optimality conditions of the arc, we must either reduce the flow on arc (p, q) to zero or change the potentials so that the reduced cost of arc (p, q) becomes zero.

We first try to reroute the flow x^*_{pq} from node p to node q without violating any of the optimality conditions. We do so by solving a maximum flow problem defined as follows: (1) set the flow on the arc (p, q) to zero, thus creating an excess of x^*_{pq} at node p and a deficit of x^*_{pq} at node q; (2) designate node p as the source node and node q as the sink node; and (3) send a maximum of x^*_{pq} units from the source to the sink. We permit the maximum flow algorithm, however, to change flows only on arcs with zero reduced costs since otherwise it would generate a solution that might violate (9.8). Let v° denote the flow sent from node p to node q and x° denote the resulting arc flow. If $v^\circ = x^*_{pq}$, then x° denotes a minimum cost flow of the modified problem. In this case the optimal objective function values of the original and modified problems are the same.

On the other hand, if $v^\circ < x^*_{pq}$, the maximum flow algorithm yields an s–t cut $[S, \bar{S}]$ with the properties that $p \in S$, $q \in \bar{S}$, and every forward arc in the cut with zero reduced cost has flow equal to its capacity and every backward arc in the cut with zero reduced cost has zero flow. We then decrease the node potential of every node in $\bar{S}$ by 1 unit. It is easy to verify by case analysis that this change in node potentials maintains the complementary slackness optimality conditions and, furthermore, decreases the reduced cost of arc (p, q) to zero. Consequently, we can set the flow on arc (p, q) equal to $x^*_{pq} - v^\circ$ and obtain a feasible minimum cost flow. In this case the objective function value of the modified problem is $x^*_{pq} - v^\circ$ units more than that of the original problem.

9.12 SUMMARY

The minimum cost flow problem is the central object of study in this book. In this chapter we began our study of this important class of problems by showing how minimum cost flow problems arise in several application settings and by considering

Algorithm	Number of iterations	Features
Cycle-canceling algorithm	$O(mCU)$	1. Maintains a feasible flow x at every iteration and augments flows along negative cycles in $G(x)$. 2. At each iteration, solves a shortest path problem with arbitrary arc lengths to identify a negative cycle. 3. Very flexible: some rules for selecting negative cycles leads to polynomial-time algorithms.
Successive shortest path algorithm	$O(nU)$	1. Maintains a pseudoflow x satisfying the optimality conditions and augments flow along shortest paths from excess nodes to deficit nodes in $G(x)$. 2. At each iteration, solves a shortest path problem with nonnegative arc lengths. 3. Very flexible: by selecting augmentations carefully, we can obtain several polynomial-time algorithms.
Primal–dual algorithm	$O(\min\{nU, nC\})$	1. Maintains a pseudoflow x satisfying the optimality conditions. Solves a shortest path problem to update node potentials and attempts to reduce primal infeasibility by the maximum amount by solving a maximum flow problem. 2. At each iteration, solves both a shortest path problem with nonnegative arc lengths and a maximum flow problem. 3. Closely related to the successive shortest path algorithm: instead of sending flow along one shortest path, sends flow along all shortest paths.
Out-of-kilter algorithm	$O(nU)$	1. Maintains a feasible flow x at each iteration and attempts to satisfy the optimality conditions by augmenting flows along shortest paths. 2. At each iteration, solves a shortest path problem with nonnegative arc lengths. 3. Can be generalized to solve situations in which the flow x maintained by the algorithm might not satisfy the flow bounds on the arcs.
Relaxation algorithm	See Exercise 9.27	1. Somewhat different from other minimum cost flow algorithms. 2. Maintains a pseudoflow x satisfying the optimality conditions and modifies arc flows and node potentials so that a Lagrangian objective function does not decrease and occasionally increases. 3. With the incorporation of some heuristics, the algorithm is very efficient in practice and yields the fastest available algorithm for some classes of minimum cost flow problems.

Figure 9.20 Summary of pseudopolynomial-time algorithms for the minimum cost flow problem.

the simplest pseudopolynomial-time algorithms for solving these problems. These pseudopolynomial-time algorithms include classical algorithms that are important because of both their historical significance and because they provide the essential building blocks and core ideas used in more efficient algorithms. Our algorithmic development relies heavily upon optimality conditions for the minimum cost flow problem that we developed and proved in the following equivalent frameworks: negative cycle optimality conditions, reduced cost optimality conditions, and complementary slackness optimality conditions. The negative cycle optimality conditions state that a feasible flow x is an optimal flow if and only if the residual network $G(x)$ contains no negative cycle. The reduced cost optimality conditions state that a feasible flow x is an optimal flow if and only if the reduced cost of each arc in the residual network is nonnegative. The complementary slackness optimality conditions are adaptations of the linear programming optimality conditions for network flows. As part of this general discussion in this chapter, we also examined minimum cost flow duality.

We developed several minimum cost flow algorithms: the cycle-canceling, successive shortest path, primal–dual, out-of-kilter, and relaxation algorithms. These algorithms represent a good spectrum of approaches for solving the same problem: Some of these algorithms maintain primal feasible solutions and strive toward optimality; others maintain primal infeasible solutions that satisfy the optimality conditions and strive toward feasibility. These algorithms have some commonalties as well—they all repeatedly solve shortest path problems. In fact, in Exercises 9.57 and 9.58 we establish a very strong result by showing that the cycle-canceling, successive shortest path, primal–dual, and out-of-kilter algorithms are all equivalent in the sense that if initialized properly, they perform the same sequence of augmentations. Figure 9.20 summarizes the basic features of the algorithms discussed in this chapter.

Finally, we discussed sensitivity analysis for the minimum cost flow problem. We showed how to reoptimize the minimum cost flow problem, after we have made unit changes in the supply/demand vector or the arc capacities, by solving a shortest path problem, and how to handle unit changes in the cost vector by solving a maximum flow problem. Needless to say, these reoptimization procedures are substantially faster than solving the problem afresh if the changes in the problem data are sufficiently small.

REFERENCE NOTES

In this chapter and in these reference notes we focus on pseudopolynomial-time nonsimplex algorithms for solving minimum cost flow problems. In Chapter 10 we provide references for polynomial-time minimum cost flow algorithms, and in Chapter 11 we give references for simplex-based algorithms.

Ford and Fulkerson [1957] developed the primal–dual algorithms for the capacitated transportation problem; Ford and Fulkerson [1962] later generalized this approach for solving the minimum cost flow problem. Jewell [1958], Iri [1960], and Busaker and Gowen [1961] independently developed the successive shortest path algorithm. These researchers showed how to solve the minimum cost flow problem as a sequence of shortest path problems with arbitrary arc lengths. Tomizava [1972]

and Edmonds and Karp [1972] independently observed that if the computations use node potentials, it is possible to implement these algorithms so that the shortest path problems have nonnegative arc lengths.

Minty [1960] and Fulkerson [1961b] independently developed the out-of-kilter algorithm. Aashtiani and Magnanti [1976] have described an efficient implementation of this algorithm. The description of the out-of-kilter algorithm presented in Section 9.9 differs substantially from the development found in other textbooks. Our description is substantially shorter and simpler because it avoids tedious case analyses. Moreover, our description explicitly highlights the use of Dijkstra's algorithm; because other descriptions do not focus on the shortest path computations, they find an accurate worst-case analysis of the algorithm much more difficult to conduct.

The cycle-canceling algorithm is credited to Klein [1967]. Three special implementations of the cycle-canceling algorithms run in polynomial time: the first, due to Barahona and Tardos [1989] (which, in turn, modifies an algorithm by Weintraub [1974]), augments flow along (negative) cycles with the maximum possible improvement; the second, due to Goldberg and Tarjan [1988], augments flow along minimum mean cost (negative) cycles; and the third, due to Wallacher and Zimmerman [1991], augments flow along minimum ratio cycles.

Zadeh [1973a,1973b] described families of minimum cost flow problems on which each of several algorithms—the cycle-canceling algorithm, successive shortest path algorithm, primal–dual algorithm, and out-of-kilter algorithm—perform an exponential number of iterations. The fact that the same families of networks are bad for many network algorithms suggests an interrelationship among the algorithms. The insightful paper by Zadeh [1979] points out that each of the algorithms we have just mentioned are indeed equivalent in the sense that they perform the same sequence of augmentations, which they obtained through shortest path computations, provided that we initialize them properly and break ties using the same rule.

Bertsekas and Tseng [1988b] developed the relaxation algorithm and conducted extensive computational investigations of it. A FORTRAN code of the relaxation algorithm appears in Bertsekas and Tseng [1988a]. Their study and those conducted by Grigoriadis [1986] and Kennington and Wang [1990] indicate that the relaxation algorithm and the network simplex algorithm (described in Chapter 11) are the two fastest available algorithms for solving the minimum cost flow problem in practice. When the supplies/demands at nodes are relatively small, the successive shortest path algorithm is the fastest algorithm. Previous computational studies conducted by Glover, Karney, and Klingman [1974] and Bradley, Brown, and Graves [1977] have indicated that the network simplex algorithm is consistently superior to the primal–dual and out-of-kilter algorithms. Most of these computational testings have been done on random network flow problems generated by the well-known computer program NETGEN, suggested by Klingman, Napier, and Stutz [1974].

The applications of the minimum cost flow problem that we discussed Section 9.2 have been adapted from the following papers:

1. Distribution problems (Glover and Klingman [1976])
2. Reconstructing the left ventricle from x-ray projections (Slump and Gerbrands [1982])
3. Racial balancing of schools (Belford and Ratliff [1972])

4. Optimal loading of a hopping airplane (Gupta [1985] and Lawania [1990])
5. Scheduling with deferral costs (Lawler [1964])
6. Linear programming with consecutive 1's in columns (Veinott and Wagner [1962])

Elsewhere in this book we describe other applications of the minimum cost flow problem. These applications include (1) leveling mountainous terrain (Application 1.4, Farley [1980]), (2) the forest scheduling problem (Exercise 1.10), (3) the entrepreneur's problem (Exercise 9.1, Prager [1957]), (4) vehicle fleet planning (Exercise 9.2), (5) optimal storage policy for libraries (Exercise 9.3, Evans [1984]), (6) zoned warehousing (Exercise 9.4, Evans [1984]), (7) allocation of contractors to public works (Exercise 9.5, Cheshire, McKinnon, and Williams [1984]), (8) phasing out capital equipment (Exercise 9.6, Daniel [1973]), (9) the terminal assignment problem (Exercise 9.7, Esau and Williams [1966]), (10) linear programs with consecutive or circular 1's in rows (Exercises 9.8 and 9.9, Bartholdi, Orlin, and Ratliff [1980]), (11) capacitated maximum spanning trees (Exercise 9.54, Garey and Johnson [1979]), (12) fractional b-matching (Exercise 9.55), (13) the nurse scheduling problem (Exercise 11.1), (14) the caterer problem (Exercise 11.2, Jacobs [1954]), (15) project assignment (Exercise 11.3), (16) passenger routing (Exercise 11.4), (17) allocating receivers to transmitters (Exercise 11.5, Dantzig [1962]), (18) faculty–course assignment (Exercise 11.6, Mulvey [1979]), (19) optimal rounding of a matrix (Exercise 11.7, Bacharach [1966], Cox and Ernst [1982]), (20) automatic karotyping of chromosomes (Application 19.8, Tso, Kleinschmidt, Mitterreiter, and Graham [1991]), (21) just-in-time scheduling (Application 19.10, Elmaghraby [1978], Levner and Nemirovsky [1991]), (22) time–cost trade-off in project management (Application 19.11, Fulkerson [1961a] and Kelly [1961]), (23) models for building evacuation (Application 19.13, Chalmet, Francis and Saunders [1982]), (24) the directed Chinese postman problem (Application 19.14, Edmonds and Johnson [1973]), (25) warehouse layout (Application 19.17, Francis and White [1976]), (26) rectilinear distance facility location (Application 19.18, Cabot, Francis, and Stary [1970]), (27) dynamic lot sizing (Application 19.19, Zangwill [1969]), (28) multistage production-inventory planning (Application 19.23, Evans [1977]), (29) mold allocation (Application 19.24, Love and Vemuganti [1978]), (30) a parking model (Exercise 19.17, Dirickx and Jennergren [1975]), (31) the network interdiction problem (Exercise 19.18, Fulkerson and Harding [1977]), (32) truck scheduling (Exercises 19.19 and 19.20, Gavish and Schweitzer [1974]), and (33) optimal deployment of firefighting companies (Exercise 19.21, Denardo, Rothblum, and Swersey [1988]).

The applications of the minimum cost flow problems are so vast that we have not been able to describe many other applications in this book. The following list provides a set of references to some other applications: (1) warehousing and distribution of a seasonal product (Jewell [1957]), (2) economic distribution of coal supplies in the gas industry (Berrisford [1960]), (3) upsets in round-robin tournaments (Fulkerson [1965]), (4) optimal container inventory and routing (Horn [1971]), (5) distribution of empty rail containers (White [1972]), (6) optimal defense of a network (Picard and Ratliff [1973]), (7) telephone operator scheduling (Segal [1974]), (8) multifacility minimax location problem with rectilinear distances (Dearing and Francis [1974]), (9) cash management problems (Srinivasan [1974]), (10) multiproduct mul-

tifacility production-inventory planning (Dorsey, Hodgson, and Ratliff [1975]), (11) "hub" and "wheel" scheduling problems (Arisawa and Elmaghraby [1977]), (12) the warehouse leasing problem (Lowe, Francis, and Reinhardt [1979]), (13) multiattribute marketing models (Srinivasan [1979]), (14) material handling systems (Maxwell and Wilson [1981]), (15) microdata file merging (Barr and Turner [1981]), (16) determining service districts (Larson and Odoni [1981]), (17) control of forest fires (Kourtz [1984]), (18) allocating blood to hospitals from a central blood bank (Sapountzis [1984]), (19) market equilibrium problems (Dafermos and Nagurney [1984]), (20) automatic chromosome classifications (Tso [1986]), (21) the city traffic congestion problem (Zawack and Thompson [1987]), (22) satellite scheduling (Servi [1989]), and (23) determining k disjoint cuts in a network (Wagner [1990]).

EXERCISES

9.1. Enterpreneur's problem (Prager [1957]). An entrepreneur faces the following problem. In each of T periods, he can buy, sell, or hold for later sale some commodity, subject to the following constraints. In each period i he can buy at most α_i units of the commodity, can holdover at most β_i units of the commodity for the next period, and must sell at least γ_i units (perhaps due to prior agreements). The enterpreneur cannot sell the commodity in the same period in which he buys it. Assuming that p_i, w_i, and s_i denote the purchase cost, inventory carrying cost, and selling price per unit in period i, what buy–sell policy should the entreprenuer adopt to maximize total profit in the T periods? Formulate this problem as a minimum cost flow problem for $T = 4$.

9.2. Vehicle fleet planning. The Millersburg Supply Company uses a large fleet of vehicles which it leases from manufacturers. The company has forecast the following pattern of vehicle requirements for the next 6 months:

Month	Jan.	Feb.	Mar.	Apr.	May	June
Vehicles required	430	410	440	390	425	450

Millersburg can lease vehicles from several manufacturers at various costs and for various lengths of time. Three of the plans appear to be the best available: a 3-month lease for \$1700; a 4-month lease for \$2200; and a 5-month lease for \$2600. The company can undertake a lease beginning in any month. On January 1 the company has 200 cars on lease, all of which go off lease at the end of February. Formulate the problem of determining the most economical leasing policy as a minimum cost flow problem. (*Hint*: Observe that the linear (integer) programming formulation of this problem has consecutive 1's in each column. Then use the result in Application 9.6.)

9.3. Optimal storage policy for libraries (Evans [1984]). A library facing insufficient primary storage space for its collection is considering the possibility of using secondary facilities, such as closed stacks or remote locations, to store portions of its collection. These options are preferred to an expensive expansion of primary storage. Each secondary storage facility has limited capacity and a particular access costs for retrieving information. Through appropriate data collection, we can determine the usage rates for the information needs of the users. Let b_j denote the capacity of storage facility j and v_j

denote the access cost per unit item from this facility. In addition, let a_i denote the number of items of a particular class i requiring storage and let u_i denote the expected rate (per unit time) that we will need to retrieve books from this class. Our goal is to store the books in a way that will minimize the expected retrieval cost.

(a) Show how to formulate the problem of determining an optimal policy as a transportation problem. What is the special structure of this problem? Transportation problems with this structure have become known as *factored transportation problems*.

(b) Show that the simple rule that repeatedly assigns items with the greatest retrievel rate to the storage facility with lowest access cost specifies an optimal solution of this library storage problem.

9.4. Zoned warehousing (Evans [1984]). In the storage of multiple, say p, items in a zoned warehouse, we need to extract (pick) items in large quantities (perhaps by pallet loads). Suppose that the warehouse is partitioned into q zones, each with a different distance to the shipping area. Let B_j denote the storage capacity of zone j and let d_j denote the average distance from zone j to the shipping area. For each item i, we know (1) the space requirement per unit (r_i), (2) the average order size in some common volume unit (s_i), and (3) the average number of orders per day (f_i). The problem is to determine the quantity of each item to allocate to each zone in order to minimize the average daily handling costs. Assume that the handling cost is linearly proportional to the distance and to the volume moved.

(a) Formulate this problem as a factored transportation problem (as defined in Exercise 9.3).

(b) Specify a simple rule that yields an optimal solution of the zoned warehousing problem.

9.5. Allocation of contractors to public works (Cheshire, McKinnon, and Williams [1984]). A large publicly owned corporation has 12 divisions in Great Britain. Each division faces a similar problem. Each year the division subcontracts work to private contractors. The work is of several different types and is done by teams, each of which is capable of doing all types of work. One of these divisions is divided into several districts: the jth district requires r_j teams. The contractors are of two types: experienced and inexperienced. Each contractor i quotes a price c_{ij} to have a team conduct the work in district j. The objective is to allocate the work in the districts to the various contractors, satisfying the following conditions: (1) each district j has r_j assigned teams; (2) the division contracts with contractor i for no more than u_i teams, the maximum number of teams it can supply; and (3) each district has at least one experienced contractor assigned to it. Formulate this problem as a minimum cost flow problem for a division with three districts, and with two experienced and two inexperienced contractors. (*Hint*: Split each district node into two nodes, one of which requires an experienced contractor.)

9.6. Phasing out capital equipment (Daniel [1973]). A shipping company wants to phase out a fleet of (homogeneous) general cargo ships over a period of p years. Its objective is to maximize its cash assets at the end of the p years by considering the possibility of prematurely selling ships and temporary replacing them by charter ships. The company faces a known nonincreasing demand for ships. Let $d(i)$ denote the demand of ships in year i. Each ship earns a revenue of r_k units in period k. At the beginning of year k, the company can sell any ship that it owns, accruing a cash inflow of s_k dollars. If the company does not own sufficiently many ships to meet its demand, it must hire additional charter ships. Let h_k denote the cost of hiring a ship for the kth year. The shipping company wants to meet its commitments and at the same time maximize the cash assets at the end of the pth year. Formulate this problem as a minimum cost flow problem.

9.7. Terminal assignment problem (Esau and Williams [1966]). Centralized teleprocessing networks often contain many (as many as tens of thousands) relatively unsophisticated geographically dispersed terminals. These terminals need to be connected to a central processor unit (CPU) either by direct lines or though *concentrators*. Each concentrator is connected to the CPU through a high-speed, cost-effective line that is capable of merging data flow streams from different terminals and sending them to the CPU. Suppose that the concentrators are in place and that each concentrator can handle at most K terminals. For each terminal j, let c_{oj} denote the cost of laying down a direct line from the CPU to the terminal and let c_{ij} denote the line construction cost for connecting concentrator i to terminal j. The decision problem is to construct the minimum cost network for connecting the terminals to the CPU. Formulate this problem as a minimum cost flow problem.

9.8. Linear programs with consecutive 1's in rows. In Application 9.6 we considered linear programs with consecutive 1's in each column and showed how to transform them into minimum cost flow problems. In this and the next exercise we study several related linear programming problems and show how we can solve them by solving minimum cost flow problems. In this exercise we study linear programs with consecutive 1's in the rows. Consider the following (integer) linear program with consecutive 1's in the rows:

$$\text{Minimize} \quad c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4$$

subject to

$$\begin{aligned} x_2 + x_3 + x_4 &\geq 20 \\ x_1 + x_2 + x_3 + x_4 &\geq 30 \\ x_2 + x_3 &\geq 15 \\ x_3 + x_4 &\geq 10 \\ x_1, x_2, x_3, x_4 &\geq 0 \text{ and integer.} \end{aligned}$$

Transform this problem to a minimum cost flow problem. (*Hint*: Use the same transformation of variables that we used in Application 4.6.)

9.9. Linear programs with circular 1's in rows (Bartholdi, Orlin, and Ratliff [1980]). In this exercise we consider a generalization of Exercise 9.8 with the 1's in each row arranged consecutively when we view columns in the wraparound fashion (i.e., we consider the first column as next to the last column). A special case of this problem is the telephone operator scheduling problem that we discussed in Application 4.6. In this exercise we focus on the telephone operator scheduling problem; nevertheless, the approach easily extends to any general linear program with circular 1's in the rows. We consider a version of the telephone operator scheduling in which we incur a cost c_i whenever an operator works in the ith shift, and we wish to satisfy the minimum operator requirement for each hour of the day at the least possible cost. We can formulate this "cyclic staff scheduling problem" as the following (integer) linear program.

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

subject to

$$\begin{aligned} y_{i-7} + y_{i-6} + \cdots + y_i &\geq b(i) \quad \text{for all } i = 7 \text{ to } 23, \\ y_{17+i} + \cdots + y_{23} + y_0 + \cdots + y_i &\geq b(i) \quad \text{for all } i = 0 \text{ to } 6, \\ y_i &\geq 0 \quad \text{for all } i = 1 \text{ to } 23. \end{aligned}$$

(a) For a parameter p, let $\mathcal{P}(p)$ denote the cyclic staff scheduling problem when we impose the additional constraint $\sum_{i=0}^{23} y_i = p$, and let $z(p)$ denote the optimal objective value of this problem. Show how to transform $\mathcal{P}(p)$, for a fixed value of p, into a minimum cost flow problem. (*Hint*: Use the same transformation of variables that we used in Application 4.6 and observe that each row has one $+1$ and one -1. Then use the result of Theorem 9.9.)

(b) Show that $z(p)$ is a (piecewise linear) convex function of p. (*Hint*: Show that if y' is an optimal solution of $\mathcal{P}(p')$ and y'' is an optimal solution of $\mathcal{P}(p'')$, then for any weighting parameter λ, $0 \leq \lambda \leq 1$, the point $\lambda y' + (1 - \lambda)y''$ is a feasible solution of $\mathcal{P}(\lambda p' + (1 - \lambda)p'')$.)

(c) In the cyclic staff scheduling problem, we wish to determine a value of p, say p^*, satisfying the property that $z(p^*) \leq z(p)$ for all feasible p. Show how to solve the cyclic staff scheduling problem in polynomial time by performing binary search on the values of p. (*Hint*: For any integer p, show how to determine whether $p \leq p^*$ by solving problems $\mathcal{P}(p)$ and $\mathcal{P}(p + 1)$.)

9.10. Racial balancing of schools. In this exercise we discuss some generalizations of the problem of racial balancing of schools that we described in Application 9.3. Describe how would you modify the formulation to include the following additional restrictions (consider each restriction separately).

(a) We prohibit the assignment of a student from location i to school j if the travel distance d_{ij} between these location exceeds some specified distance D.

(b) We include the distance traveled between location i and school j in the objective function only if d_{ij} is greater than some specified distance D' (e.g., we account for the distance traveled only if a student needs to be bussed).

(c) We impose lower and upper bounds on the number of black students from location i who are assigned to school j.

9.11. Show how to transform the equipment replacement problem described in Application 9.6 into a shortest path problem. Give the resulting formulation for $n = 4$.

9.12. This exercise is based on the equipment replacement problem that we discussed in Application 9.6.

(a) The problem as described allows us to buy and sell the equipment only yearly. How would you model the situation if you could make decisions every half year?

(b) How sensitive do you think the optimal solution would be to the length T of planning period? Can you anticipate a situation in which the optimal replacement plan would change drastically if we were to increase the length of the planning period to $T + 1$?

9.13. Justify the minimum cost flow formulation that we described in Application 9.4 for the problem of optimally loading a hopping airplane. Establish a one-to-one correspondence between feasible passenger routings and feasible flows in the minimum cost flow formulation of the problem.

9.14. In this exercise we consider one generalization of the tanker scheduling problem discussed in Application 6.6. Suppose that we can compute the profit associated with each available shipment (depending on the revenues and the operating cost directly attributable to that shipment). Let the profits associated with the shipments 1, 2, 3, and 4 be 10, 10, 3, and 4, respectively. In addition to the operating cost, we incur a fixed charge of 5 units to bring a ship into service. We want to determine the shipments we should make and the number of ships to use to maximize net profits. (Note that it is not necessary to honor all possible shipping commitments.) Formulate this problem as a minimum cost flow problem.

9.15. Consider the following data, with $n = 4$, for the employment scheduling problem that we discussed in Application 9.6. Formulate this problem as a minimum cost flow problem and solve it by the successive shortest path algorithm.

$\{c_{ij}\} =$

	1	2	3	4	5
1	—	20	35	50	55
2	—	—	15	30	40
3	—	—	—	25	35
4	—	—	—	—	10

i	1	2	3	4
$d(i)$	20	15	30	25

9.16. Figure 9.21(b) shows the optimal solution of the minimum cost flow problem shown in Figure 9.21(a). First, verify that x^* is a feasible flow.
(a) Draw the residual network $G(x^*)$ and show that it contains no negative cycle.
(b) Specify a set of node potentials π that together with x^* satisfy the reduced cost optimality conditions. List each arc in the residual network and its reduced cost.

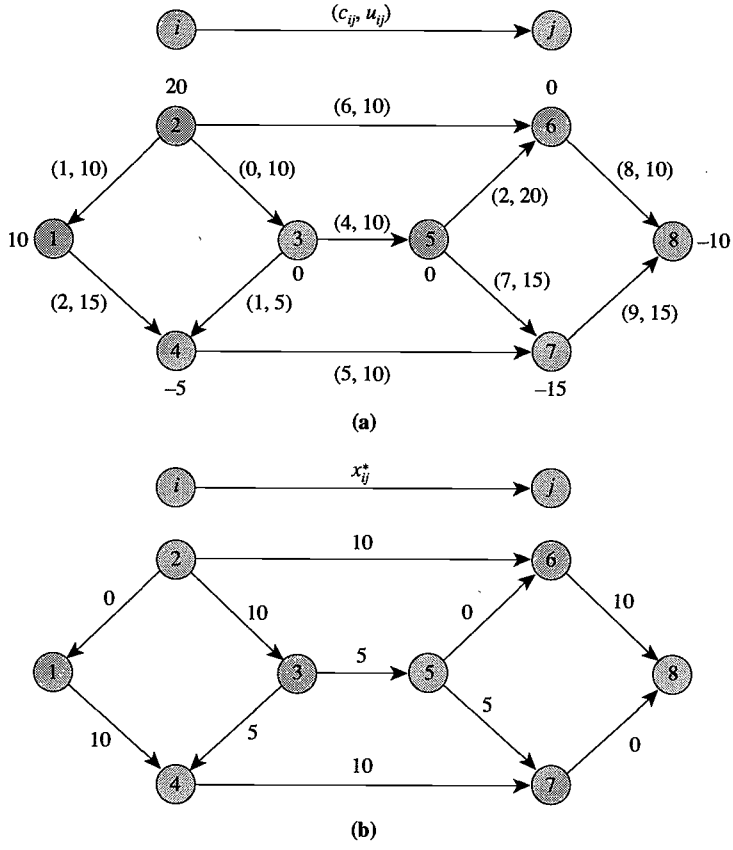


Figure 9.21 Minimum cost flow problem: (a) problem data; (b) optimal solution.

(c) Verify that the solution x^* satisfies the complementary slackness optimality conditions. To do so, specify a set of optimal node potentials and list the reduced cost of each arc in A.

9.17. (a) Figure 9.22(a) gives the data and an optimal solution for a minimum cost flow problem. Assume that all arcs are uncapacitated. Determine optimal node potentials.

(b) Consider the uncapacitated minimum cost flow problem shown in Figure 9.22(b). For this problem the vector $\pi = (0, -6, -9, -12, -5, -8, -15)$ is an optimal set of node potentials. Determine an optimal flow in the network.

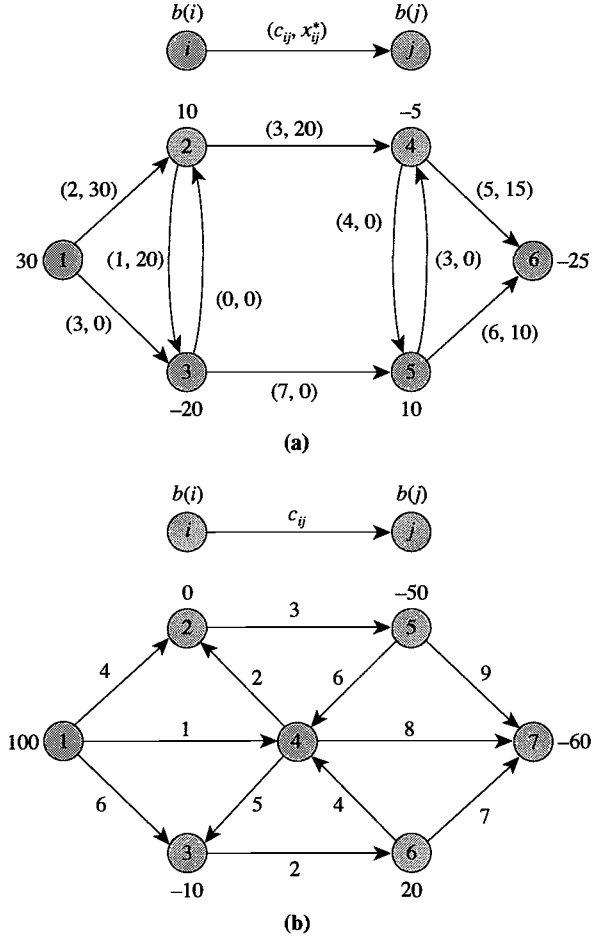


Figure 9.22 Example for Exercise 9.17.

9.18. Solve the problem shown in Figure 9.23 by the cycle-canceling algorithm. Use the zero flow as the starting solution.

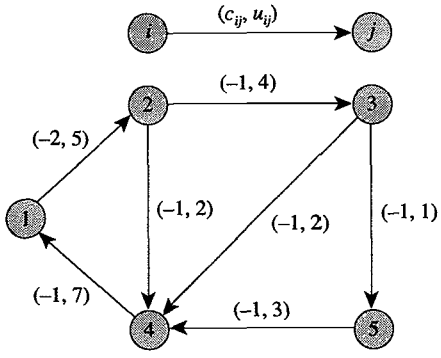


Figure 9.23 Example for Exercise 9.18.

9.19. Show that if we apply the cycle-canceling algorithm to the minimum cost flow problem shown in Figure 9.24, some sequence of augmentations requires 2×10^6 iterations to solve the problem.

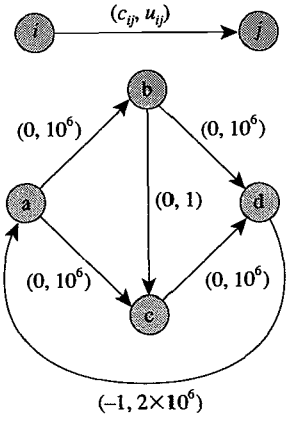


Figure 9.24 Network where cycle canceling algorithm performs 2×10^6 iterations.

9.20. Apply the successive shortest path algorithm to the minimum cost flow problem shown in Figure 9.25. Show that the algorithm performs eight augmentations, each of unit flow, and that the cost of these augmentations (i.e., sum of the arc costs in the path

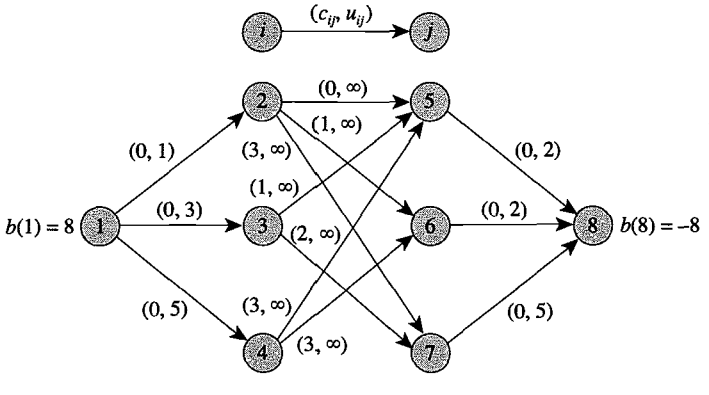


Figure 9.25 Example for Exercise 9.20.

in the residual network) is 0, 1, 2, 3, 3, 4, 5, and 6. How many iterations does the primal–dual algorithm require to solve this problem?

9.21. Construct a class of minimum cost flow problems for which the number of iterations performed by the successive shortest path algorithm might grow exponentially in log U. (*Hint*: Consider the example shown in Figure 9.24.)

9.22. Figure 9.26 specifies the data and a feasible solution for a minimum cost flow problem. With respect to zero node potentials, list the in-kilter and out-of-kilter arcs. Apply the out-of-kilter algorithm to find an optimal flow in the network.

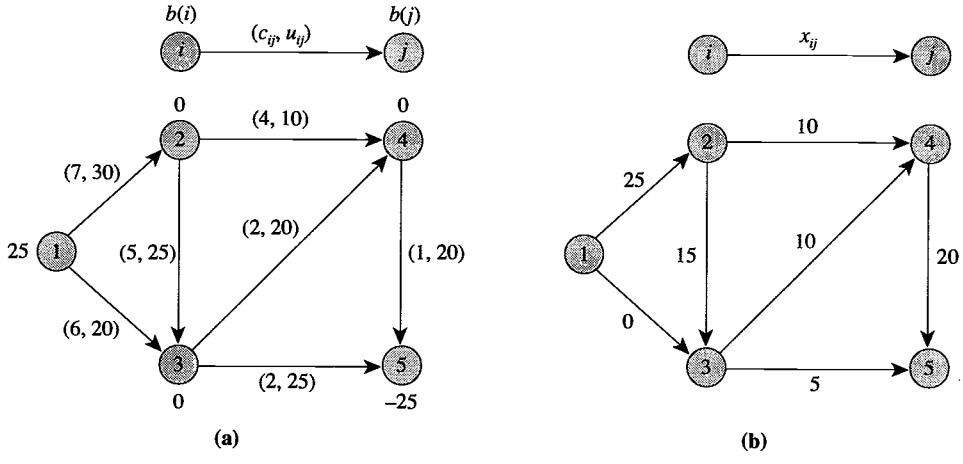


Figure 9.26 Example for Exercises 9.22 and 9.23: (a) problem data; (b) feasible flow.

9.23. Consider the minimum cost flow problem shown in Figure 9.26. Starting with zero pseudoflow and zero node potentials, apply the relaxation algorithm to establish an optimal flow.

9.24. Figure 9.21(b) specifies an optimal solution for the minimum cost flow problem shown in Figure 9.21(a). Reoptimize the solution with respect to the following changes in the problem data: (1) when c_{23} increases from 0 to 6; (2) when c_{78} decreases from 9 to 2; (3) when $b(2)$ decreases to 15 and $b(8)$ increases to -5; and (4) when u_{23} increases to 20. Treat these changes individually.

9.25. Assuming that we set one node potential to value zero, show that nC is an upper bound and that $-nC$ is a lower bound on the optimal value of any node potential.

9.26. Justify the out-of-kilter algorithm described in Section 9.9 for the case when arcs can violate their flow bounds. Show that in the execution of the algorithm, the kilter number of arcs are nonincreasing and at least one kilter number strictly decreases at every iteration.

9.27. Obtain a worst-case bound on the total number of iterations performed by the relaxation algorithm. Compare this bound with the number of iterations performed by the cycle-canceling, successive shortest path, and primal–dual algorithms.

9.28. Show that if the pair (x, π) satisfies the complementary slackness optimality conditions (9.8), it also satisfies the reduced cost optimality conditions (9.7).

9.29. Prove that if x^* is an optimal flow and π is an optimal set of node potentials, the pair (x^*, π) satisfies the complementary slackness optimality conditions. In your proof, do not use the strong duality theorem. (*Hint*: Suppose that the pair (x, π) satisfies the optimality conditions for some flow x. Show that $c^{\pi}(x^* - x) = 0$ and use this fact to prove the desired result.)

9.30. With respect to an optimal solution x^* of a minimum cost flow problem, suppose that we redefine arc capacities u' as follows:

$$u'_{ij} = \begin{cases} u_{ij} & \text{if } x^*_{ij} = u_{ij} \\ \infty & \text{if } x^*_{ij} < u_{ij}. \end{cases}$$

Show that x^* is also an optimal solution of the minimum cost flow problem with the arc capacities as u'.

9.31. With respect to an optimal solution x^* of a minimum cost flow problem, suppose that we redefine arc capacities $u' = x^*$. Show that x^* is also an optimal solution of the minimum cost flow problem with arc capacities u'.

9.32. In Section 2.4 we showed how to transform a minimum cost flow problem in an undirected network in which all lower bounds are zero into a minimum cost flow problem in a directed network. Explain why this approach does not work when some lower bounds on arc flows exceed zero.

9.33. In the minimum cost flow problem, suppose that one specified arc (p, q) has no lower and upper flow bounds. How would you transform this problem into the standard minimum cost flow problem?

9.34. As we have seen in Section 2.4, the uncapacitated transportation problem is equivalent to the minimum cost flow problem in the sense that we can always transform either problem into a version of another problem. If we can solve the uncapacitated transportation problem in $O(g(n, m))$ time, can we also solve the minimum cost flow problem in $O(g(n, m))$ time?

9.35. In the *min-cost max-flow problem* defined on a directed network $G = (N, A)$, we wish to send the maximum amount of flow from a node s to a node t at the minimum possible total cost. That is, among all maximum flows, find the one with the smallest cost.

(a) Show how to formulate any minimum cost flow problem as a min-cost max-flow problem.

(b) Show how to convert any min-cost max-flow problem into a circulation problem.

9.36. Suppose that in a minimum cost flow problem, some arcs have infinite capacities and some arc costs are negative. (Assume that the lower bounds on all arc flows are zero.)

(a) Show that the minimum cost flow problem has a finite optimal solution if and only if the uncapacitated arcs do not contain a negative cost-directed cycle.

(b) Let B denote the sum of the finite arc capacities and the supplies $b(\cdot)$ of all the supply nodes. Show that the minimum cost flow problem always has an optimal solution in which each arc flow is at most B. Conclude that without any loss of generality, we can assume that in the minimum cost flow problem (with a bounded optimal solution value) every arc is capacitated. (*Hint*: Use the flow decomposition property.)

9.37. Suppose that in a minimum cost flow problem, some arcs have infinite capacities and some arc costs are negative. Let B denote the sum of the finite arc capacities and the right-hand-side coefficients $b(i)$ for all the supply nodes. Let z and z' denote the objective function values of the minimum cost flow problem when we set the capacity of each infinite capacity arc to the value B and $B + 1$, respectively. Show that the objective function of the minimum cost flow problem is unbounded if and only if $z' < z$.

9.38. In a minimum cost flow network, suppose that in addition to arc capacities, nodes have upper bounds imposed upon the entering flow. Let $w(i)$ be the maximum flow that can enter node $i \in N$. How would you solve this generalization of the minimum cost flow problem?

9.39. Let (k, l) and (p, q) denote a minimum cost arc and a maximum cost arc in a network. Is it possible that no minimum cost flow have a positive flow on arc (k, l)? Is it possible that every minimum cost flow have a positive flow on arc (p, q)? Justify your answers.

9.40. Prove or disprove the following claims.

(a) Suppose that all supply/demands and arc capacities in a minimum cost flow problem

are all even integers. Then for some optimal flow x^*, each arc flow x_{ij}^* is an even number.

(b) Suppose that all supply/demands and arc capacities in a minimum cost circulation problem are all even integers. Then for some optimal flow x^*, each arc flow x_{ij}^* is an even number.

9.41. Let x^* be an optimal solution of the minimum cost flow problem. Define G° as a subgraph of the residual network $G(x^*)$ consisting of all arcs with zero reduced cost. Show that the minimum cost flow problem has an alternative optimal solution if and only if G° contains a directed cycle.

9.42. Suppose that you are given a nonintegral optimal solution to a minimum cost flow problem with integral data. Suggest a method for converting this solution into an integer optimal solution. Your method should maintain optimality of the solution at every step.

9.43. Suppose that the pair (x, π), for some pseudoflow x and some node potentials π, satisfies the reduced cost optimality conditions. Define $G^\circ(x)$ as a subgraph of the residual network $G(x)$ consisting of only those arcs with zero residual capacity. Define the cost of an arc (i, j) in $G^\circ(x)$ as c_{ij} if $(i, j) \in A$, and as $-c_{ij}$ otherwise. Show that every directed path in $G^\circ(x)$ between any pair of nodes is a shortest path in $G(x)$ between the same pair of nodes with respect to the arc costs c_{ij}.

9.44. Let x^1 and x^2 be two distinct (alternate) minimum cost flows in a network. Suppose that for some arc (k, l), $x_{kl}^1 = p$, $x_{kl}^2 = q$, and $p < q$. Show that for every $0 \le \lambda \le 1$, the minimum cost flow problem has an optimal solution x (possibly, noninteger) with $x_{kl} = (1 - \lambda)p + \lambda q$.

9.45. Let π^1 and π^2 be two distinct (alternate) optimal node potentials of a minimum cost flow problem. Suppose that for some node k, $\pi^1(k) = p$, $\pi^2(k) = q$, and $p < q$. Show that for every $0 \le \lambda \le 1$, the minimum cost flow problem has an optimal set of node potentials π (possibly, noninteger) with $\pi(k) = (1 - \lambda)p + \lambda q$.

9.46. (a) In the transportation problem, does adding a constant k to the cost of every outgoing arc from a specified supply node affect the optimality of a given optimal solution? Would adding a constant k to the cost of every incoming arc to a specified demand node affect the optimality of a given optimal solution?

(b) Would your answers to the questions in part (a) be the same if they were posed for the minimum cost flow problem instead of the transportation problem?

9.47. In Section 9.7 we described the following practical improvement of the successive shortest path algorithm: (1) terminate the execution of Dijkstra's algorithm whenever it permanently labels a deficit node l, and (2) modify the node potentials by setting $\pi(i)$ to $\pi(i) - d(i)$ if node i is permanently labeled; and by setting $\pi(i)$ to $\pi(i) - d(l)$ if node i is temporarily labeled. Show that after the algorithm has updated the node potentials in this manner, all the arcs in the residual network have nonnegative reduced costs and all the arcs in the shortest path from node k to node l have zero reduced costs. (*Hint*: Use the result in Exercise 5.9.)

9.48. Would multiplying each arc cost in a network by a constant k change the set of optimal solutions of the minimum cost flow problem? Would adding a constant k to each arc cost change the set of optimal solutions?

9.49. In Section 9.11 we described a method for performing sensitivity analysis when we increase the capacity of an arc (p, q) by 1 unit. Modify the method to perform the analysis when we decrease the capacity of the arc (p, q) by 1 unit.

9.50. In Section 9.11 we described a method for performing sensitivity analysis when we increase the cost of an arc (p, q) by 1 unit. Modify the method to perform the analysis when we decrease the cost of the arc (p, q) by 1 unit.

9.51. Suppose that we have to solve a minimum cost flow problem in which the sum of the supplies exceeds the sum of the demands, so we need to retain some of the supply at some nodes. We refer to this problem as the *minimum cost flow problem with surplus*. Specify a linear programming formulation of this problem. Also show how to transform this problem into an (ordinary) minimum cost flow problem.

9.52. This exercise concerns the minimum cost flow problem with surplus, as defined in Exercise 9.51. Suppose that we have an optimal solution of a minimum cost flow problem with surplus and we increase the supply of some node by 1 unit, holding the other data fixed. Show that the optimal objective function value cannot increase, but it might decrease. Show that if we increase the demand of a node by 1 unit, holding the other data fixed, the optimal objective function value cannot decrease, but it might increase.

9.53. More-for-less paradox (Charnes and Klingman [1971]). The more-for-less paradox shows that it is possible to send *more* flow from the supply nodes to the demand nodes of a minimum cost flow problem at *lower* cost even if all arc costs are nonnegative. To establish this more-for-less paradox, consider the minimum cost flow problem shown in Figure 9.27. Assume that all arc capacities are infinite.

(a) Show that the solution given by $x_{14} = 11$, $x_{16} = 9$, $x_{25} = 2$, $x_{26} = 8$, $x_{35} = 11$, and $x_{37} = 14$, is an optimal flow for this minimum cost flow problem. What is the total cost of flow?

(b) Suppose that we increase the supply of node 2 by 2 units, increase the demand of node 4 by 2 units, and reoptimize the solution using the method described in Section 9.11. Show that the total cost of flow decreases.

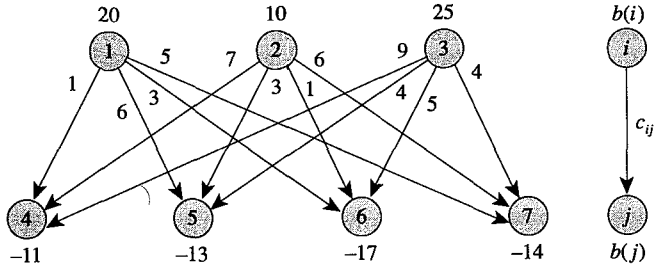


Figure 9.27 More-for-less paradox.

9.54. Capacitated minimum spanning tree problem (Garey and Johnson [1979]). In a complete undirected network with arc lengths c_{ij} and a specially designated node s, called the *central site*, we associate an integer requirement r_i with every node $i \in N - \{s\}$. In the capacitated minimum spanning tree problem, we want to identify a minimum cost spanning tree so that when we send flow on this tree from the central site to the other nodes to satisfy their flow requirements, no tree arc has a flow of more than a given arc capacity R, which is the same for all arcs. Show that when each r_i is 0 or 1, and $R = 1$, we can solve this problem as a minimum cost flow problem. (*Hint*: Model this problem as a minimum cost flow problem with node capacities, as discussed in Exercise 9.38.)

9.55. Fractional *b*-matching problem. Let $G = (N, A)$ be an undirected graph in which each node $i \in N$ has an associated supply $b(i)$ and each arc $(i, j) \in A$ has an associated cost c_{ij} and capacity u_{ij}. In the *b-matching problem*, we wish to find a minimum cost subgraph of G with exactly b arcs incident to every node. The *fractional b-matching problem* is a relaxation of the b-matching problem and can be stated as the following linear program:

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

subject to

$$\sum_{j\in A(i)} x_{ij} = b(i) \qquad \text{for all } i \in N,$$

$$0 \le x_{ij} \le u_{ij} \qquad \text{for all } (i, j) \in A.$$

We assume that $x_{ij} = x_{ji}$ for every arc $(i, j) \in A$. We can define a related minimum cost flow problem as follows. Construct a bipartite network $G' = (N' \cup N'', A')$ with $N' = \{1', 2', \ldots, n'\}$, $N'' = \{1'', 2'', \ldots, n''\}$, $b(i') = b(i)$, and $b(i'') = -b(i)$. For each arc $(i, j) \in A$, the network G' contains two associated arcs (i', j'') and (j', i''), each with cost c_{ij} and capacity u_{ij}.

(a) Show how to transform every solution x of the fractional b-matching problem with cost z into a solution x' of the minimum cost flow problem with cost $2z$. Similarly, show that if x' is a solution of the minimum cost flow problem with cost z', then $x_{ij} = (x'_{ij} + x'_{ji})/2$ is a feasible solution of the fractional b-matching problem with cost $z'/2$. Use these results to show how to solve the fractional b-matching problem.

(b) Show that the fractional b-matching problem always has an optimal solution in which each arc flow x_{ij} is a multiple of $\frac{1}{2}$. Also show that if all the supplies and the capacities are even integers, the fractional b-matching problem always has an integer optimal solution.

9.56. Bottleneck transportation problem. Consider a transportation problem with a traversal time τ_{ij} instead of a cost c_{ij} associated with each arc (i, j). In the *bottleneck transportation problem* we wish to satisfy the requirements of the demand nodes from the supply nodes in the least time possible [i.e., we wish to find a flow x that minimizes the quantity $\max\{\tau_{ij}: (i, j) \in A \text{ and } x_{ij} > 0\}$].

(a) Suggest an application of the bottleneck transportation problem.

(b) Suppose that we arrange the arc traversal times in the nondecreasing order of their values. Let $\tau_1 < \tau_2 < \cdots < \tau_l$ be the distinct values of the arc traversal times (thus $l \leq m$). Let *FS(k, found)* denote a subroutine that finds whether the transportation problem has a feasible solution using only those the arcs with traversal times less than or equal to τ_k; assume that the subroutine assigns a value true/false to found. Suggest a method for implementing the subroutine FS(k, found).

(c) Using the subroutine FS(k, found), write a pseudocode for solving the bottleneck transportation problem.

9.57. Equivalence of minimum cost flow algorithms (Zadeh [1979])

(a) Apply the successive shortest path algorithm to the minimum cost flow problem shown in Figure 9.28. Show that it performs four augmentations from node 1 to node 6, each of unit flow.

(b) Add the arc (1, 6) with sufficiently large cost and with $u_{16} = 4$ to the example in part (a). Observe that setting $x_{16} = 4$ and $x_{ij} = 0$ for all other arcs gives a feasible flow in the network. With this flow as the initial flow, apply the cycle-canceling algorithm and always augment flow along a negative cycle with minimum cost. Show that this algorithm also performs four unit flow augmentations from node 1 to node 6 along the same paths as in part (a) and in the same order, except that the flow returns to node 1 through the arc (6, 1) in the residual network.

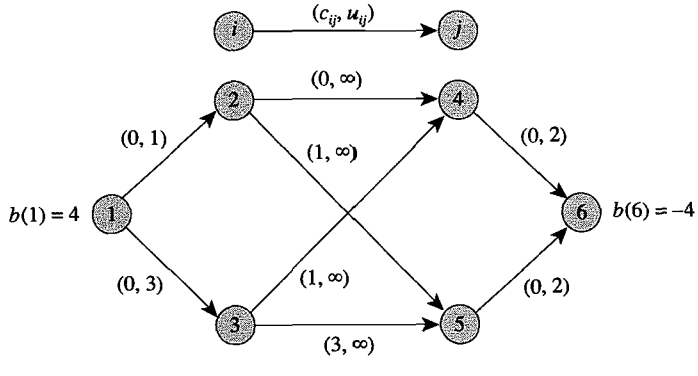


Figure 9.28 Equivalence of minimum cost flow algorithms.

(c) Using parts (a) and (b) as background, prove the general result that if initialized properly, the successive shortest path algorithm and the cycle-canceling algorithm (with augmentation along a most negative cycle) are equivalent in the sense that they perform the same augmentations and in the same order.

9.58. Modify and initialize the minimum cost flow problem in Figure 9.28 appropriately so that when we apply the out-of-kilter algorithm to this problem, it also performs four augmentation in the same order as the successive shortest path algorithm. Then prove the equivalence of the out-of-kilter algorithm with the successive shortest path algorithm in general.

10

MINIMUM COST FLOWS: POLYNOMIAL ALGORITHMS

Success generally depends upon knowing how long it takes to succeed.
—Montesquieu

Chapter Outline

10.1 INTRODUCTION

In Chapter 9 we studied several different algorithms for solving minimum cost problems. Although these algorithms guarantee finite convergence whenever the problem data are integral, the computations are not bounded by any polynomial in the underlying problem specification. In the spirit of computational complexity theory, this situation is not completely satisfactory: It does not provide us with any good theoretical assurance that the algorithms will perform well on all problems that we might encounter. The circumstances are quite analogous to our previous development of maximum flow algorithms; we started by first developing straightforward, but not necessarily polynomial, algorithms for solving those problems, and then enhanced these algorithms by changing the algorithmic strategy and/or by using clever data structures and implementations. This situation raises the following natural questions: (1) Can we devise algorithms that are polynomial in the usual problem parameters: number n of nodes, number m of arcs, log U (the log of the largest supply/demand or arc capacity), and log C (the log of the largest cost coefficient), and (2) can we develop strongly polynomial-time algorithms (i.e., algorithms whose running time depends upon only on n and m)? A strongly polynomial-time algorithm has one important theoretical advantage: It will solve problems with irrational data.

In this chapter we provide affirmative answers to these questions. To develop polynomial-time algorithms, we use ideas that are similar to those we have used before: namely, scaling of the capacity data and/or of the cost data. We consider

three polynomial-time algorithms: (1) a capacity scaling algorithm that is a scaled version of the successive shortest path algorithm that we discussed in Chapter 9, (2) a cost scaling algorithm that is a generalization of the preflow-push algorithm for the maximum flow problem, and (3) a double scaling algorithm that simultaneously scales both the arc capacities and the costs.

Scaling is a powerful idea that has produced algorithmic improvements to many problems in combinatorial optimization. We might view scaling algorithms as follows. We start with the optimality conditions for the network flow problem we are examining, but instead of enforcing these conditions exactly, we generate an "approximate" solution that is permitted to violate one (or more) of the conditions by an amount Δ. Initially, by choosing Δ quite large, for example as C or U, we will easily be able to find a starting solution that satisfies the relaxed optimality conditions. We then reset the parameter Δ to $\Delta/2$ and reoptimize so that the approximate solution now violates the optimality conditions by an amount of at most $\Delta/2$. We then repeat the procedure, reoptimizing again until the approximate solution violates the conditions by an amount of at most $\Delta/4$, and so on. This solution strategy is quite flexible and leads to different algorithms depending on which of the optimality conditions we relax and how we perform the reoptimizations.

Our discussion of the capacity scaling algorithm for the maximum flow problem in Section 7.3 provides one example. A feasible flow to the maximum flow problem is optimal if the residual network contains no augmenting path. In the capacity scaling algorithm, we relaxed this condition so that after the Δ-scaling phase, the residual network can contain an augmenting path, but only if its capacity were less than Δ. The excess scaling algorithm for the maximum flow problem provides us with another example. In this case the residual network again contains no path from the source node s to the sink node t; however, at the end of the Δ-scaling phase, we relaxed a feasibility requirement requiring that the flow into every node other than the source and sink equals the flow out of that node. Instead, we permitted the excess at each node to be as large as Δ during the Δ-scaling phase.

In this chapter, by applying a scaling approach to the algorithms that we considered in Chapter 9, we develop polynomial-time versions of these algorithms. We begin by developing a modified version of the successive shortest path algorithm in which we relax two optimality conditions in the Δ-scaling phase: (1) We permit the solution to violate supply/demand constraints by an amount Δ, and (2) we permit the residual network to contain negative cost cycles. The resulting algorithm reduces the number of shortest path computations from nU to $m \log U$.

We next describe a cost-scaling algorithm that uses another concept of approximate optimality; at the end of each ϵ-scaling phase (ϵ plays the role of Δ) we obtain a feasible flow that satisfies the property that the reduced cost of each arc in the residual network is greater than or equal to $-\epsilon$ (instead of zero). To find the optimal solution during the ϵ-scaling phase, this algorithm carries out a sequence of push and relabel operations that are similar to the preflow-push algorithm for maximum flows. The generic cost scaling algorithm runs in $O(n^2 m \log(nC))$ time. We also describe a special "wave implementation" of this algorithm that chooses nodes for the push/relabel operations in a specific order. This specialization requires $O(n^3 \log(nC))$ time.

We then describe a *double scaling algorithm* that combines the features of both cost and capacity scaling. This algorithm works with two nested loops. In the outer loop we scale the costs, and in the inner loop we scale the capacities. Introducing capacity scaling as an inner loop within a cost scaling approach permits us to find augmenting paths very efficiently. This resulting double scaling algorithm solves the minimum cost flow problem in $O(nm \log U \log(nC))$ time.

All of these algorithms require polynomial time; they are not, however, strongly polynomial time because their time bounds depend on $\log U$ and/or $\log C$. Developing strongly polynomial-time algorithms seems to require a somewhat different approach. Although most strongly polynomial-time algorithms use ideas of data scaling, they also use another idea: By invoking the optimality conditions, they are able to show that at intermediate steps of the algorithm, they have already discovered part of the optimal solution (e.g., optimal flow), so that they are able to reduce the problem size. In Sections 10.5, 10.6, and 10.7 we consider three different strongly polynomial-time algorithms whose analysis invokes this "problem reduction argument."

In Section 10.5 we analyze the minimum mean cycle-canceling algorithm that we described in Section 9.6. Recall that this algorithm augments flow at each step on a cycle with the smallest average cost, averaged over the number of arcs in the cycle, until the residual network contains no negative cost cycle; at this point, the current flow is optimal. As we show in this section, we can view this algorithm as finding a sequence of improved approximately optimal solutions (in the sense that the reduced cost of every arc is greater than or equal to $-\epsilon$, with ϵ decreasing throughout the algorithm). This algorithm has the property that if the magnitude of the reduced cost of any arc is sufficiently large (as a function of ϵ), the flow on that arc remains fixed at its upper or lower bound throughout the remainder of the algorithm and so has this value in the optimal solution. This property permits us to show that the algorithm fixes the flow on an arc and does so sufficiently often so that we obtain an $O(n^2m^3 \log n)$ time algorithm for the capacitated minimum cost flow problem. One interesting characteristic of this algorithm is that it does not explicitly monitor ϵ or explicitly fix the flow variables. These features of the algorithm are by-products of the analysis.

The strongly polynomial-time algorithm that we consider in Section 10.6 solves the linear programming dual of the minimum cost flow problem. This *repeated capacity scaling algorithm* is a variant of the capacity scaling algorithm that we discuss in Section 10.2. This algorithm uses a scaling parameter Δ as in the capacity scaling algorithm, but shows that periodically the flow on some arc (i, j) becomes sufficiently large (as a function of Δ), at which point we are able to reduce the size of the dual linear program by one, which is equivalent to *contraction* in the primal network. This observation permits us to reduce the size of the problem successively by contracting nodes. The end result is an algorithm requiring $O((m^2 \log n)(m + n \log n))$ time for the minimum cost flow problem.

In Section 10.7 we consider an *enhanced scaling algorithm* that is a hybrid version of the capacity scaling algorithm and the repeated capacity scaling algorithm. By choosing a scaling parameter Δ carefully and by permitting a somewhat broader choice of the augmenting paths at each step, this algorithm is able to fix variables more quickly than the repeated capacity scaling algorithm. As a consequence, it

solves fewer shortest path problems and solves capacitated minimum cost flow problems in $O((m \log n)(m + n \log n))$ time, which is currently the best known polynomial-time bound for solving the capacitated minimum cost flow problem.

10.2 CAPACITY SCALING ALGORITHM

In Chapter 9 we considered the successive shortest path algorithm, one of the fundamental algorithms for solving the minimum cost flow problem. An inherent drawback of this algorithm is that its augmentations might carry relatively small amounts of flow, resulting in a fairly large number of augmentations in the worst case. By incorporating a scaling technique, the capacity algorithm described in this section guarantees that each augmentation carries *sufficiently large* flow and thereby reduces the number of augmentations substantially. This method permits us to improve the worst-case algorithmic performance from $O(nU \cdot S(n, m, nC))$ to $O(m \log U \cdot S(n, m, nC))$. [Recall that U is an upper bound on the largest supply/demand and largest capacity in the network, and $S(n, m, C)$ is the time required to solve a shortest path problem with n nodes, m arcs, and nonnegative costs whose values are no more than C. The reason that the running time involves $S(n, m, nC)$ rather than $S(n, m, C)$ is that the costs in the residual network are reduced costs, and the reduced cost of an arc could be as large as nC.]

The capacity scaling algorithm is a variant of the successive shortest path algorithm. It is related to the successive shortest path algorithm, just as the capacity scaling algorithm for the maximum flow problem (discussed in Section 7.3) is related to the labeling algorithm (discussed in Section 6.5). Recall that the labeling algorithm performs $O(nU)$ augmentations; by sending flows along paths with *sufficiently large* residual capacities, the capacity scaling algorithm reduces the number of augmentations to $O(m \log U)$. In a similar fashion, the capacity scaling algorithm for the minimum cost flow problem ensures that each shortest path augmentation carries a sufficiently large amount of flow; this modification to the algorithm reduces the number of successive shortest path iterations from $O(nU)$ to $O(m \log U)$. This algorithm not only improves on the algorithmic performance of the successive shortest path algorithm, but also illustrates how small changes in an algorithm can produce significant algorithmic improvements (at least in the worst case).

The capacity scaling algorithm applies to the general capacitated minimum cost flow problem. It uses a pseudoflow x and the imbalances $e(i)$ as defined in Section 9.7. The algorithm maintains a pseudoflow satisfying the reduced cost optimality condition and gradually converts this pseudoflow into a flow by identifying shortest paths from nodes with excesses to nodes with deficits and augmenting flows along these paths. It performs a number of scaling phases for different values of a parameter Δ. We refer to a scaling phase with a specific value of Δ as the *Δ-scaling phase*. Initially, $\Delta = 2^{\lfloor \log U \rfloor}$. The algorithm ensures that in the Δ-scaling phase each augmentation carries exactly Δ units of flow. When it is not possible to do so because no node has an excess of at least Δ, or no node has a deficit of at least Δ, the algorithm reduces the value of Δ by a factor of 2 and repeats the process. Eventually, $\Delta = 1$ and at the end of this scaling phase, the solution becomes a flow. This flow must be an optimal flow because it satisfies the reduced cost optimality condition.

For a given value of Δ, we define two sets $S(\Delta)$ and $T(\Delta)$ as follows:

$$S(\Delta) = \{i : e(i) \geq \Delta\},$$

$$T(\Delta) = \{i : e(i) \leq -\Delta\}.$$

In the Δ-scaling phase, each augmentation must start at a node in $S(\Delta)$ and end at a node in $T(\Delta)$. Moreover, the augmentation must take place on a path along which every arc has residual capacity of at least Δ. Therefore, we introduce another definition: The Δ-*residual network* $G(x, \Delta)$ is defined as the subgraph of $G(x)$ consisting of those arcs whose residual capacity is at least Δ. In the Δ-scaling phase, the algorithm augments flow from a node in $S(\Delta)$ to a node in $T(\Delta)$ along a shortest path in $G(x, \Delta)$. The algorithm satisfies the property that every arc in $G(x, \Delta)$ satisfies the reduced cost optimality condition; however, arcs in $G(x)$ but not in $G(x, \Delta)$ might violate the reduced cost optimality condition. Figure 10.1 presents an algorithmic description of the capacity scaling algorithm.

Notice that the capacity scaling algorithm augments exactly Δ units of flow in the Δ-scaling phase, even though it could augment more. For uncapacitated problems, this tactic leads to the useful property that all arc flows are always an integral multiple of Δ. (Why might capacitated networks not satisfy this property?) Several variations of the capacity scaling algorithm discussed in Sections 10.5 and 14.5 adopt the same tactic.

To establish the correctness of the capacity scaling algorithm, observe that the 2Δ-scaling phase ends when $S(2\Delta) = \phi$ or $T(2\Delta) = \phi$. At that point, either $e(i) < 2\Delta$ for all $i \in N$ or $e(i) > -2\Delta$ for all $i \in N$. These conditions imply that the sum of the excesses (whose magnitude equals the sum of deficits) is bounded by $2n\Delta$.

```
algorithm capacity scaling;
begin
    x := 0 and π := 0;
    Δ := 2^⌊log U⌋;
    while Δ ≥ 1
    begin {Δ-scaling phase}
        for every arc (i, j) in the residual network G(x) do
            if r_ij ≥ Δ and c^π_ij < 0 then send r_ij units of flow along arc (i, j),
              update x and the imbalances e(·);
        S(Δ) := {i ∈ N : e(i) ≥ Δ};
        T(Δ) := {i ∈ N : e(i) ≤ −Δ};
        while S(Δ) ≠ ∅ and T(Δ) ≠ ∅ do
        begin
            select a node k ∈ S(Δ) and a node l ∈ T(Δ);
            determine shortest path distances d(·) from node k to all other nodes in the
                Δ-residual network G(x, Δ) with respect to the reduced costs c^π_ij;
            let P denote shortest path from node k to node l in G(x, Δ);
            update π := π − d;
            augment Δ units of flow along the path P;
            update x, S(Δ), T(Δ), and G(x, Δ);
        end;
        Δ := Δ/2;
    end;
end;
```

Figure 10.1 Capacity scaling algorithm.

At the beginning of the Δ-scaling phase, the algorithm first checks whether every arc (i, j) in Δ-residual network satisfies the reduced cost optimality condition $c_{ij}^{\pi} \geq 0$. The arcs introduced in the Δ-residual network at the beginning of the Δ-scaling phase [i.e., those arcs (i, j) for which $\Delta \leq r_{ij} < 2\Delta$] might not satisfy the optimality condition (since, conceivably, $c_{ij}^{\pi} < 0$). Therefore, the algorithm immediately saturates those arcs (i, j) so that they drop out of the residual network; since the reversal of these arcs (j, i) satisfy the condition $c_{ji}^{\pi} = -c_{ij}^{\pi} > 0$, they satisfy the optimality condition. Notice that because $r_{ij} < 2\Delta$, saturating any such arc (i, j) changes the imbalance of its endpoints by at most 2Δ. As a result, after we have saturated all the arcs violating the reduced cost optimality condition, the sum of the excesses is bounded by $2n\Delta + 2m\Delta = 2(n + m)\Delta$.

In the Δ-scaling phase, each augmentation starts at a node $k \in S(\Delta)$, terminates at a node $l \in T(\Delta)$, and carries at least Δ units of flow. Note that Assumption 9.4 implies that the Δ-residual network contains a directed path from node k to node l, so we always succeed in identifying a shortest path from node k to node l. Augmenting flow along a shortest path in $G(x, \Delta)$ preserves the property that every arc satisfies the reduced cost optimality condition (see Section 9.3). When either $S(\Delta)$ or $T(\Delta)$ is empty, the Δ-scaling phase ends. At this point we divide Δ by a factor of 2 and start a new scaling phase. Within $O(\log U)$ scaling phases, $\Delta = 1$, and by the integrality of data, every node imbalance will be zero at the end of this phase. In this phase $G(x, \Delta) \equiv G(x)$ and every arc in the residual network satisfies the reduced cost optimality condition. Consequently, the algorithm will obtain a minimum cost flow at the end of this scaling phase.

As we have seen, the capacity scaling algorithm is easy to state. Similarly, its running time is easy to analyze. We have noted previously that in the Δ-scaling phase the sum of the excesses is bounded by $2(n + m)\Delta$. Since each augmentation in this phase carries at least Δ units of flow from a node in $S(\Delta)$ to a node in $T(\Delta)$, each augmentation reduces the sum of the excesses by at least Δ units. Therefore, a scaling phase can perform at most $2(n + m)$ augmentations. Since we need to solve a shortest path problem to identify each augmenting path, we have established the following result.

Theorem 10.1. *The capacity scaling algorithm solves the minimum cost flow problem in $O(m \log U\, S(n, m, nC))$ time.* ◆

10.3 COST SCALING ALGORITHM

In this section we describe a cost scaling algorithm for the minimum cost flow problem. This algorithm can be viewed as a generalization of the preflow-push algorithm for the maximum flow problem; in fact, the algorithm reveals an interesting relationship between the maximum flow and minimum cost flow problems. This algorithm relies on the concept of approximate optimality.

Approximate Optimality

A flow x or a pseudoflow x is said to be *ϵ-optimal* for some $\epsilon > 0$ if for some node potentials π, the pair (x, π) satisfies the following *ϵ-optimality conditions*:

$$\text{If } c_{ij}^{\pi} > \epsilon, \text{ then } x_{ij} = 0. \tag{10.1a}$$

$$\text{If } -\epsilon \le c_{ij}^{\pi} \le \epsilon, \text{ then } 0 \le x_{ij} \le u_{ij}. \tag{10.1b}$$

$$\text{If } c_{ij}^{\pi} < -\epsilon, \text{ then } x_{ij} = u_{ij}. \tag{10.1c}$$

These conditions are relaxations of the (exact) complementary slackness optimality conditions (9.8) that we discussed in Section 9.3; note that these conditions reduce to the complementary slackness optimality conditions when $\epsilon = 0$. The *exact* optimality conditions (9.8) imply that any combination of (x_{ij}, c_{ij}^{π}) lying on the thick lines shown in Figure 10.2(a) is optimal. The ϵ-optimality conditions (10.1) imply that any combination of (x_{ij}, c_{ij}^{π}) lying on the thick lines or in the hatched region in Figure 10.2(b) is ϵ-optimal.

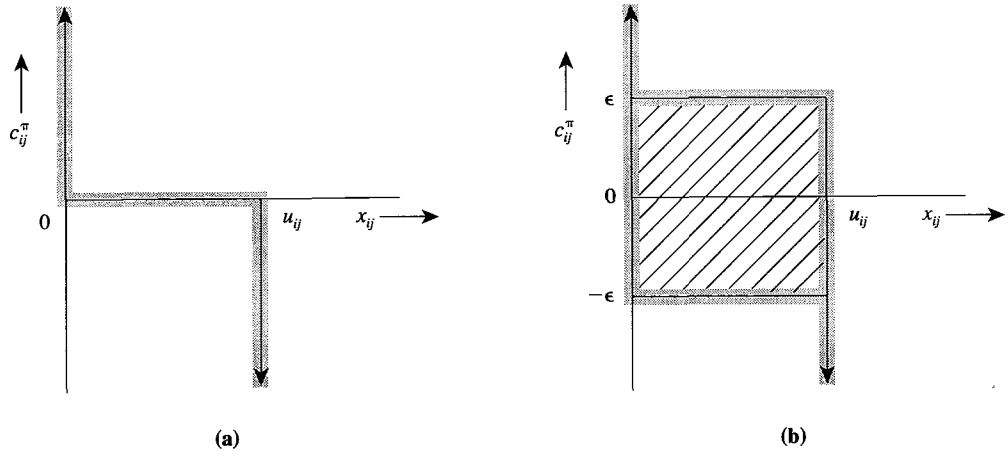


Figure 10.2 Illustrating the optimality condition for arc (i, j): (a) exact optimality condition for arc (i, j); (b) ϵ-optimality condition for arc (i, j).

The ϵ-optimality conditions assume the following simpler form when stated in terms of the residual network $G(x)$: A flow x or a pseudoflow x is said to be *ϵ-optimal* for some $\epsilon > 0$ if x, together with some node potential vector π, satisfies the following *ϵ-optimality conditions* (we leave the proof as an exercise for the reader):

$$c_{ij}^{\pi} \ge -\epsilon \quad \text{for every arc } (i, j) \text{ in the residual network } G(x). \tag{10.2}$$

Lemma 10.2. *For a minimum cost flow problem with integer costs, any feasible flow is ϵ-optimal whenever $\epsilon \ge C$. Moreover, if $\epsilon < 1/n$, then any ϵ-optimal feasible flow is an optimal flow.*

Proof. Let x be any feasible flow and let $\pi = 0$. Then $c_{ij}^{\pi} = c_{ij} \ge -C$ for every arc (i, j) in the residual network $G(x)$. Therefore, x is ϵ-optimal for $\epsilon = C$.

Now consider an ϵ-optimal flow x with $\epsilon < 1/n$. Suppose that x is ϵ-optimal with respect to the node potentials π and that W is a directed cycle in $G(x)$. The condition (10·2) implies that $\sum_{(i,j)\in W} c_{ij}^{\pi} \ge -\epsilon n > -1$, because $\epsilon < 1/n$. The integrality of the costs implies that $\sum_{(i,j)\in W} c_{ij}^{\pi}$ is nonnegative. But notice that $\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} (c_{ij} - \pi(i) + \pi(j)) = \sum_{(i,j)\in W} c_{ij}$. Therefore, W cannot be a negative cost

cycle. Since $G(x)$ cannot contain any negative cycle, x must be optimal (from Theorem 9.1). ◆

Algorithm

The cost scaling algorithm treats ϵ as a parameter and iteratively obtains ϵ-optimal flows for successively smaller values of ϵ. Initially, $\epsilon = C$ and any feasible flow is ϵ-optimal. The algorithm then performs cost scaling phases by repeatedly applying an *improve-approximation* procedure that transforms an ϵ-optimal flow into a $\frac{1}{2}\epsilon$-optimal flow. After $1 + \lceil \log(nC) \rceil$ cost scaling phases, $\epsilon < 1/n$ and the algorithm terminates with an optimal flow. Figure 10.3 provides a more formal statement of the cost scaling algorithm.

```
algorithm cost scaling;
begin
    π : = 0 and ε : = C;
    let x be any feasible flow;
    while ε ≥ 1/n do
    begin
        improve-approximation(ε, x, π);
        ε : = ε/2;
    end;
    x is an optimal flow for the minimum cost flow problem;
end;
```

Figure 10.3 Cost scaling algorithm.

The improve-approximation procedure transforms an ϵ-optimal flow into a $\frac{1}{2}\epsilon$-optimal flow. It does so by (1) converting the ϵ-optimal flow into a $\frac{1}{2}\epsilon$-optimal pseudoflow, and (2) then gradually converting the pseudoflow into a flow while always maintaining $\frac{1}{2}\epsilon$-optimality of the solution. We refer to a node i with $e(i) > 0$ as *active* and say that an arc (i, j) in the residual network is *admissible* if $-\frac{1}{2}\epsilon \leq c_{ij}^{\pi} < 0$. The basic operation in the procedure is to select an active node i and perform pushes on admissible arcs (i, j) emanating from node i. When the network contains no admissible arc, the algorithm updates the node potential $\pi(i)$. Figure 10.4 summarizes the essential steps of the generic version of the improve-approximation procedure.

Recall that r_{ij} denotes the residual capacity of an arc (i, j) in $G(x)$. As in our earlier discussion of preflow-push algorithms for the maximum flow problem, if $\delta = r_{ij}$, we refer to the push as *saturating*; otherwise, it is *nonsaturating*. We also refer to the updating of the potential of a node as a *relabel* operation. The purpose of a relabel operation at node i is to create new admissible arcs emanating from this node.

We illustrate the basic operations of the improve-approximation procedure on a small numerical example. Consider the residual network shown in Figure 10.5(a). Let $\epsilon = 8$. The current pseudoflow is 4-optimal. Node 2 is the only active node in the network, so the algorithm selects it for push/relabel. Suppose that arc (2, 4) is the first admissible arc found. The algorithm pushes $\min\{e(2), r_{24}\} = \min\{30, 5\} = 5$ units of flow on arc (2, 4); this push saturates the arc. Next the algorithm identifies arc (2, 3) as admissible and pushes $\min\{e(2), r_{23}\} = \min\{25, 30\} = 25$ units on this arc. This push is nonsaturating; after the algorithm has performed this push, node

```
procedure improve-approximation(ε, x, π);
begin
    for every arc (i, j) ∈ A do
        if c^π_ij > 0 then x_ij := 0
        else if c^π_ij < 0 then x_ij := u_ij;
    compute node imbalances;
    while the network contains an active node do
    begin
        select an active node i;
        push/relabel(i);
    end;
end;
```

(a)

```
procedure push/relabel(i);
begin
    if G(x) contains an admissible arc (i, j) then
        push δ := min{e(i), r_ij} units of flow from node i to node j;
    else π(i) := π(i) + ε/2;
end;
```

(b)

Figure 10.4 Procedures of the cost scaling algorithm.

2 is inactive and node 3 is active. Figure 10.5(b) shows the residual network at this point.

In the next iteration, the algorithm selects node 3 for push/relabel. Since no admissible arc emanates from this node, we perform a relabel operation and increase the node's potential by $\epsilon/2 = 4$ units. This potential change decreases the reduced costs of the outgoing arcs, namely, (3, 4) and (3, 2), by 4 units and increases the reduced costs of the incoming arcs, namely (1, 3) and (2, 3), by 4 units [see Figure 10.5(c)]. The relabel operation creates an admissible arc, namely arc (3, 4), and we next perform a push of 15 units on this arc [see Figure 10.5(d)]. Since the current solution is a flow, the improve-approximation procedure terminates.

To identify admissible arcs emanating from node i, we use the same data structure used in the maximum flow algorithms described in Section 7.4. For each node i, we maintain a *current-arc* (i, j) which is the current candidate to test for admissibility. Initially, the current-arc of node i is the first arc in its arc list $A(i)$. To determine an admissible arc emanating from node i, the algorithm checks whether the node's current-arc is admissible, and if not, chooses the next arc in the arc list as the current-arc. Thus the algorithm passes through the arc list starting with the current-arc until it finds an admissible arc. If the algorithm reaches the end of the arc list without finding an admissible arc, it declares that the node has no admissible arc. At this point it relabels node i and again sets its current-arc to the first arc in $A(i)$.

We might comment on two practical improvements of the improve-approximation procedure. The algorithm, as stated, starts with $\epsilon = C$ and reduces ϵ by a factor of 2 in every scaling phase until $\epsilon = 1/n$. As a consequence, ϵ could become

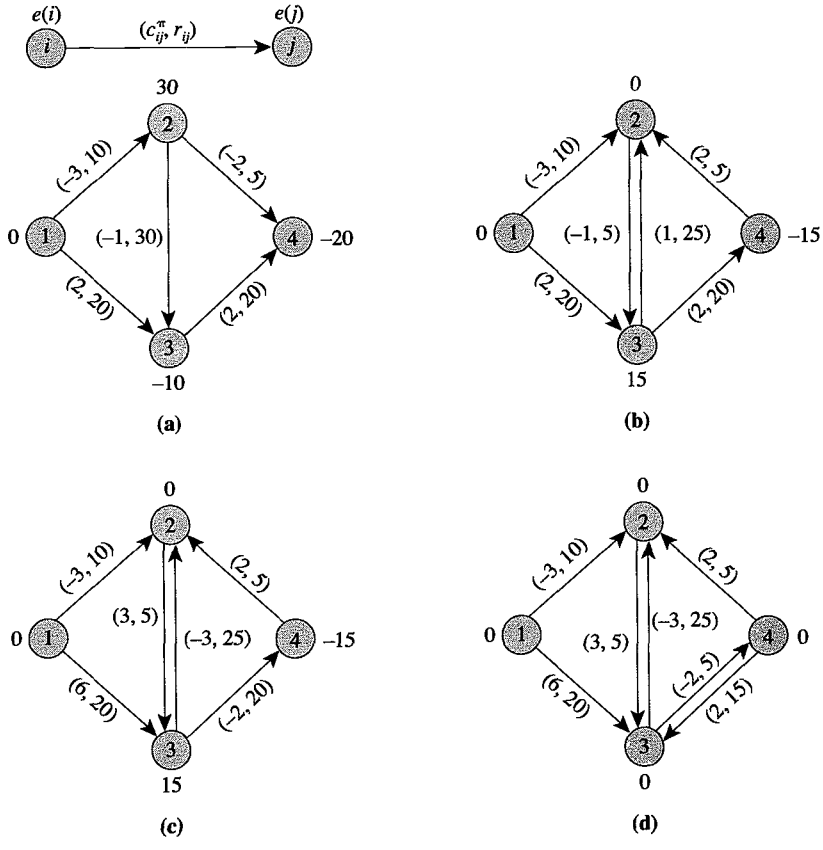


Figure 10.5 Illustration of push/relabel steps.

nonintegral during the execution of the algorithm. By slightly modifying the algorithm, however, we can ensure that ϵ remains integral. We do so by multiplying all the arc costs by n, by setting the initial value of ϵ equal to $2^{\lceil \log(nC) \rceil}$, and by terminating the algorithm when $\epsilon < 1$. It is possible to show (see Exercise 10.7) that the modified algorithm would yield an optimal flow for the minimum cost flow problem in the same computational time. Furthermore, as stated, the algorithm increases a node potential by $\epsilon/2$ during a relabel operation. As described in Exercise 10.8, we can often increase the node potential by an amount larger than $\epsilon/2$.

Analysis of the Algorithm

We show that the cost scaling algorithm correctly solves the minimum cost flow problem. In the proof, we rely on the fact that the improve-approximation procedure converts an ϵ-optimal flow into an $\epsilon/2$-optimal flow. We establish this result in the following lemma.

Lemma 10.3. *The improve-approximation procedure always maintains $\frac{1}{2}\epsilon$-optimality of the pseudoflow, and at termination yields a $\frac{1}{2}\epsilon$-optimal flow.*

Proof. We use induction on the number of pushes and relabels. At the beginning of the procedure, the algorithm sets the flows on arcs with negative reduced costs to their capacities, sets the flow on arcs with positive reduced costs to zero, and leaves the flow on arcs with zero reduced costs unchanged. The resulting pseudoflow satisfies (10.1) for $\epsilon = 0$ and thus is 0-optimal. Since a 0-optimal pseudoflow is ϵ-optimal for every ϵ, the resulting flow is also $\frac{1}{2}\epsilon$-optimal.

We next study the effect of a push on the $\frac{1}{2}\epsilon$-optimality of the solution. Pushing flow on arc (i, j) might add its reversal (j, i) to the residual network. But since $-\epsilon/2 \leq c_{ij}^{\pi} < 0$ (by the criteria of admissibility), $c_{ji}^{\pi} = -c_{ij}^{\pi} > 0$, and so this arc satisfies the $\frac{1}{2}\epsilon$-optimality condition (10.2).

What is the effect of a relabel operation? The algorithm relabels a node i when $c_{ij}^{\pi} \geq 0$ for every arc (i, j) emanating from node i in the residual network. Increasing the potential of node i by $\epsilon/2$ units decreases the reduced cost of all arcs emanating from node i by $\epsilon/2$ units. But since $c_{ij}^{\pi} \geq 0$ before the increase in π, $c_{ij}^{\pi} \geq -\epsilon/2$ after the increase, and the arc satisfies the $\frac{1}{2}\epsilon$-optimality condition. Furthermore, increasing the potential of node i by $\epsilon/2$ units increases the reduced costs of the incoming arcs at node i but maintains the $\frac{1}{2}\epsilon$-optimality condition for these arcs. These results establish the lemma. ◆

We next analyze the complexity of the improve-approximation procedure. We show that the number of relabel operations is $O(n^2)$, the number of saturating pushes is $O(nm)$, and the number of nonsaturating pushes for the generic version is $O(n^2m)$. These time bounds are comparable to those of the preflow-push algorithms for the maximum flow problem and the proof techniques are also similar. We first prove the most significant result, which bounds the number of relabel operations.

Lemma 10.4. *No node potential increases more than $3n$ times during an execution of the improve-approximation procedure.*

Proof. Let x be the current $\frac{1}{2}\epsilon$-optimal pseudoflow and x' be the ϵ-optimal flow at the end of the previous cost scaling phase. Let π and π' be the node potentials corresponding to the pseudoflow x and the flow x'. It is possible to show (see Exercise 10.9) that for every node v with an excess there exists a node w with a deficit and a sequence of nodes $v = v_0, v_1, v_2, \ldots, v_l = w$ that satisfies the property that the path $P = v_0 - v_1 - v_2 - \cdots - v_l$ is a directed path in $G(x)$ and its reversal $\overline{P} = v_l - v_{l-1} - \cdots - v_1 - v_0$ is a directed path in $G(x')$. Applying the $\frac{1}{2}\epsilon$-optimality condition to the arcs on the path P in $G(x)$, we see that

$$\sum_{(i,j)\in P} c_{ij}^{\pi} \geq -l(\epsilon/2).$$

Substituting $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ in this expression gives

$$\sum_{(i,j)\in P} c_{ij} - \pi(v) + \pi(w) \geq -l(\epsilon/2).$$

Alternatively,

$$\pi(v) \leq \pi(w) + l(\epsilon/2) + \sum_{(i,j)\in P} c_{ij}. \tag{10.3}$$

Applying the ϵ-optimality conditions to the arcs on the path $\overline{P}$ in $G(x')$, we obtain $\sum_{(j,i)\in\overline{P}} c_{ji}^{\pi'} \geq -l\epsilon$. Substituting $c_{ji}^{\pi'} = c_{ji} - \pi'(j) + \pi'(i)$ in this expression gives

$$\sum_{(j,i)\in\overline{P}} c_{ji} - \pi'(w) + \pi'(v) \geq -l\epsilon. \tag{10.4}$$

Notice that $\sum_{(j,i)\in\overline{P}} c_{ji} = -\sum_{(i,j)\in P} c_{ij}$ since $\overline{P}$ is a reversal of P. In view of this fact, we can restate (10.4) as

$$\sum_{(i,j)\in P} c_{ij} \leq l\epsilon - \pi'(w) + \pi'(v). \tag{10.5}$$

Substituting (10.5) in (10.3), we see that

$$(\pi(v) - \pi'(v)) \leq (\pi(w) - \pi'(w)) + 3l\epsilon/2. \tag{10.6}$$

Now we use the facts that (1) $\pi(w) = \pi'(w)$ (the potential of a node with negative imbalance does not change because the algorithm never selects it for push/relabel), (2) $l \leq n$, and (3) each increase in the potential increases $\pi(v)$ by at least $\epsilon/2$ units. These facts and expression (10.6) establish the lemma. ◆

Lemma 10.5. *The improve-approximation procedure performs $O(nm)$ saturating pushes.*

Proof. We show that between two consecutive saturations of an arc (i, j), the procedure must increase both the potentials $\pi(i)$ and $\pi(j)$ at least once. Consider a saturating push on arc (i, j). Since arc (i, j) is admissible at the time of the push, $c_{ij}^{\pi} < 0$. Before the algorithm can saturate this arc again, it must send some flow back from node j to node i. At that time $c_{ji}^{\pi} < 0$ or $c_{ij}^{\pi} > 0$. These conditions are possible only if the algorithm has relabeled node j. In the subsequent saturation of arc (i, j), $c_{ij}^{\pi} < 0$, which is possible only if the algorithm has relabeled node i. But by the previous lemma the improve-approximation procedure can relabel any node $O(n)$ times, so it can saturate any arc $O(n)$ times. Consequently, the number of saturating pushes is $O(nm)$. ◆

To bound the number of nonsaturating pushes, we need one more result. We define the *admissible network* of a given residual network as the network consisting solely of admissible arcs. For example, Figure 10.6(b) specifies the admissible network for the residual network given in Figure 10.6(a).

Lemma 10.6. *The admissible network is acyclic throughout the improve-approximation procedure.*

Proof. We show that the algorithm satisfies this property at every step. The result is true at the beginning of the improve-approximation procedure because the initial pseudoflow is 0-optimal and the residual network contains no admissible arc. We show that the result remains valid throughout the procedure. We always push flow on arc (i, j) with $c_{ij}^{\pi} < 0$; therefore, if the algorithm adds the reversal (j, i) of this arc to the residual network, then $c_{ji}^{\pi} > 0$ and so the reversal arc is nonadmissible. Thus pushes do not create new admissible arcs and the admissible network remains acyclic. The relabel operation at node i decreases the reduced costs of all outgoing

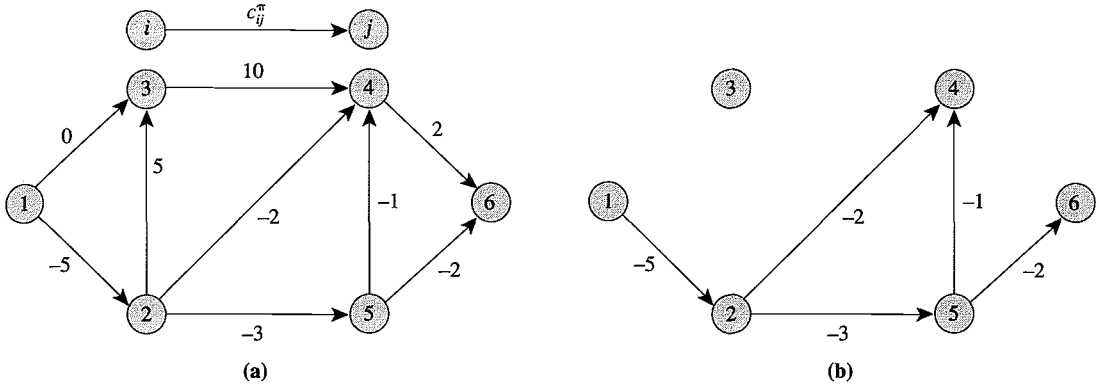


Figure 10.6 Illustration of an admissible network: (a) residual network; (b) admissible network.

arcs at node i by $\epsilon/2$ units and might create new admissible arcs. This relabel operation increases the reduced costs of all incoming arcs at node i by $\epsilon/2$ units, so all such arcs become inadmissible. Consequently, the relabel operation cannot create any directed cycle passing through node i. Thus neither of the two operations, pushes and relabels, of the algorithm can create a directed cycle, which establishes the lemma. ◆

Lemma 10.7. *The improve-approximation procedure performs $O(n^2m)$ nonsaturating pushes.*

Proof. We use a potential function argument to prove the lemma. Let $g(i)$ be the number of nodes that are reachable from node i in the admissible network and let $\Phi = \sum_{i \text{ is active}} g(i)$ be the potential function. We assume that every node is reachable from itself. For example, in the admissible network shown in Figure 10.7, nodes 1 and 4 are the only active nodes. In this network, nodes 1, 2, 3, 4, and 5 are reachable from node 1, and nodes 4 and 5 are reachable from node 4. Therefore, $g(1) = 5$, $g(4) = 2$, and $\Phi = 7$.

At the beginning of the procedure, $\Phi \leq n$ since the admissible network contains

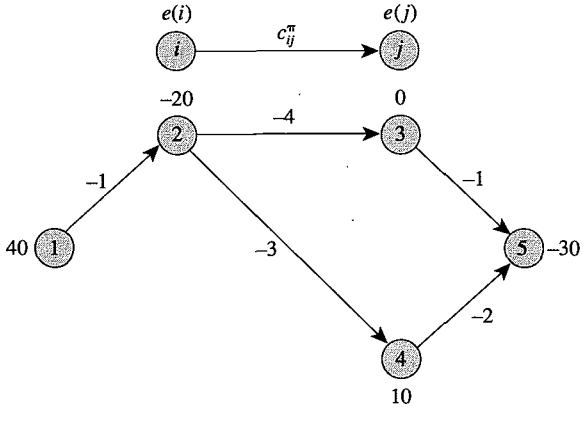


Figure 10.7 Admissible network for $\epsilon = 8$.

no arc and each $g(i) = 1$. After a saturating push on arc (i, j), node j might change its state from inactive to active, which would increase Φ by $g(j) \leq n$. Therefore, Lemma 10.5 implies that the total increase due to saturating pushes is $O(n^2m)$. A relabel operation of node i might create new admissible arcs (i, j) and will increase $g(i)$ by at most n units. But this relabel operation does not increase $g(k)$ for any other node k because it makes all incoming arcs at node k inadmissible (see the proof of Lemma 10.6). Thus the total increase due to all relabel operations is $O(n^3)$.

Finally, consider the effect on Φ of a nonsaturating push on arc (i, j). As a result of the push, node i becomes inactive and node j might change its status from inactive to active. Thus the push decreases Φ by $g(i)$ units and might increase it by another $g(j)$ units. Now notice that $g(i) \geq g(j) + 1$ because every node that is reachable from node j is also reachable from node i but node i is not reachable from node j (because the admissible network is acyclic). Therefore, a nonsaturating push decreases Φ by at least 1 unit. Consequently, the total number of nonsaturating pushes is bounded by the initial value of Φ plus the total increase in Φ throughout the algorithm, which is $O(n) + O(n^2m) + O(n^3) = O(n^2m)$. This result establishes the lemma. ◆

Let us summarize our discussion. The improve-approximation procedure requires $O(n^2m)$ time to perform nonsaturating pushes and $O(nm)$ time to perform saturating pushes. The amount of time needed to identify admissible arcs is $O(\sum_{i \in N} | A(i) | n) = O(nm)$, since between two consecutive potential increases of a node i, the algorithm will examine $| A(i) |$ arcs for testing admissibility. The algorithm could store all the active nodes in a list. Doing so would permit it to identify an active node in $O(1)$ time, so this operation would not be a bottleneck step. Consequently, the improve-approximation procedure runs in $O(n^2m)$ time. Since the cost scaling algorithm calls this procedure $1 + \lceil \log(nC) \rceil$ times, we obtain the following result.

Theorem 10.8. *The generic cost scaling algorithm runs in* $O(n^2m \log(nC))$ *time.* ◆

The cost scaling algorithm illustrates an important connection between the maximum flow and the minimum cost flow problems. Solving an improve-approximation problem is very similar to solving a maximum flow problem by the preflow-push method. Just as in the preflow-push algorithm, the bottleneck operation in the procedure is the number of nonsaturating pushes. In Chapter 7 we have seen how to reduce the number of nonsaturating pushes for the preflow-push algorithm by examining active nodes in some specific order. Similar ideas permit us to streamline the improve-approximation procedure as well. We describe one such improvement, called the *wave implementation*, that reduces the number of nonsaturating pushes from $O(n^2m)$ to $O(n^3)$.

Wave Implementation

Before we describe the wave implementation, we introduce the concept of *node examination*. In an iteration of the improve-approximation procedure, the algorithm selects a node, say node i, and either performs a saturating push or a nonsaturating

push from this node, or relabels the node. If the algorithm performs a saturating push, node i might still be active, but the algorithm might select another node in the next iteration. We shall henceforth assume that whenever the algorithm selects a node, it keeps pushing flow from that node until either its excess becomes zero or the node becomes relabeled. If we adopt this node selection strategy, the algorithm will perform several saturating pushes from a particular node followed either by a nonsaturating push or a relabel operation; we refer to this sequence of operations as a *node examination*.

The wave implementation is a special implementation of the improve-approximation procedure that selects active nodes for push/relabel steps in a specific order. The algorithm uses the fact that the admissible network is acyclic. In Section 3.4 we showed that it is always possible to order nodes of an acyclic network so that for every arc (i, j) in the network, node i occurs prior to node j. Such an ordering of nodes is called a *topological ordering*. For example, for the admissible network shown in Figure 10.6, one possible topological ordering of nodes is 1–2–5–4–3–6. In Section 3.4 we showed how to arrange the nodes of a network in a topological order in $O(m)$ time. For a given topological order, we define the *rank* of a node as n minus its number in the topological sequence. For example, in the preceding example, rank(1) = 6, rank(6) = 1 and rank(5) = 4.

Observe that each push carries flow from a node with higher rank to a node with lower rank. Also observe that pushes do not change the topological ordering of nodes since they do not create new admissible arcs. The relabel operations, however, might create new admissible arcs and consequently, might affect the topological ordering of nodes.

The wave implementation sequentially examines nodes in the topological order and if the node being examined is active, it performs push/relabel steps at the node until either the node becomes inactive or it becomes relabeled. When examined in this order, the active nodes push their excesses to nodes with lower rank, which in turn push their excesses to nodes with even lower rank, and so on. A relabel operation changes the topological order; so after each relabel operation the algorithm modifies the topological order and again starts to examine nodes according to the topological order. If within n consecutive node examinations, the algorithm performs no relabel operation, then at this point all the active nodes have discharged their excesses and the algorithm has obtained a flow. Since the algorithm performs $O(n^2)$ relabel operations, we immediately obtain a bound of $O(n^3)$ on the number of node examinations. Each node examination entails at most one nonsaturating push. Consequently, the wave algorithm performs $O(n^3)$ nonsaturating pushes per execution of improve-approximation.

To illustrate the wave implementation, we consider the pseudoflow shown in Figure 10.8. One topological order of nodes is 2–3–4–1–5–6. The algorithm first examines node 2 and pushes 20 units of flow on arc (2, 1). Then it examines node 3 and pushes 5 units of flow on arc (3, 1) and 10 units of flow on arc (3, 4). The push creates an excess of 10 units at node 4. Next the algorithm examines node 4 and sends 5 units on the arc (4, 6). Since node 4 has an excess of 5 units but has no outgoing admissible arc, we need to relabel node 4 and reexamine all nodes in the topological order starting with the first node in the order.

To complete the description of the algorithm, we need to describe a procedure

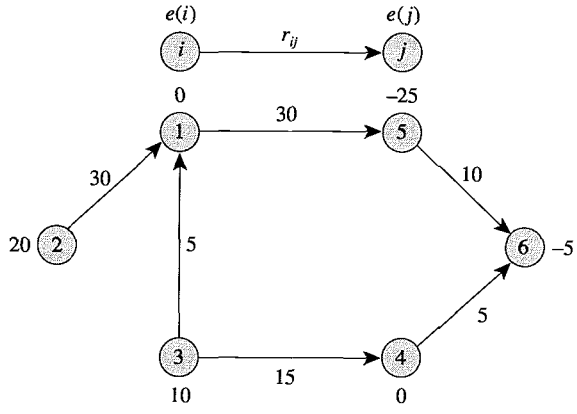


Figure 10.8 Example to illustrate the wave implementation.

for obtaining a topological order of nodes after each relabel operation. We can use an $O(m)$ algorithm to determine an initial topological ordering of the nodes (see Section 3.4). Suppose that while examining node i, the algorithm relabels this node. At this point, the network contains no incoming admissible arc at node i. We claim that if we move node i from its present position to the first position in the previous topological order leaving all other nodes intact, we obtain a topological order of the new admissible network. For example, for the admissible network given in Figure 10.8, one topological order of the nodes is 2–3–4–1–5–6. If we examine nodes in this order, the algorithm relabels node 4. After the algorithm has performed this relabel operation, the modified topological order of nodes is 4–2–3–1–5–6. This method works because (1) after the relabeling, node i has no incoming admissible arc, so assigning it to the first place in the topological order is justified; (2) the relabeling of node i might create some new outgoing admissible arcs (i, j) but since node i is first in the topological order, any such arc satisfies the conditions of a topological ordering; and (3) the rest of the admissible network does not change, so the previous order remains valid. Therefore, the algorithm maintains an ordered set of nodes (possibly as a doubly linked list) and examines nodes in this order. Whenever it relabels a node i, the algorithm moves this node to the first place in the order and again examines nodes in order starting from node i.

We have established the following result.

Theorem 10.9. *The wave implementation of the cost scaling algorithm solves the minimum cost flow problem in* $O(n^3 \log(nC))$ *time.* ◆

By examining the active nodes carefully and thereby reducing the number of nonsaturating pushes, the wave implementation improves the running time of the generic implementation of the improve-approximation procedure from $O(n^2m)$ to $O(n^3)$. A complementary approach for improving the running time is to use cleverer data structure to reduce the time per nonsaturating push. Using the *dynamic tree* data structures described in Section 8.5, we can improve the running time of the generic implementation to $O(nm \log n)$ and of the wave implementation to $O(nm \log(n^2/m))$. The references cited at the end of the chapter contain the details of these implementations.

10.4 DOUBLE SCALING ALGORITHM

As we have seen in the preceding two sections, by scaling either the arc capacities or the cost coefficients of a minimum cost flow problem, we can devise algorithms with improved worst-case performance. This development raises a natural question: Can we combine ideas from these algorithms to obtain even further improvements that are not obtained by either technique alone? In this section we provide an affirmative answer to this question. The double scaling algorithm we describe solves the capacitated minimum cost flow problem in $O(nm \log U \log(nC))$ time. When implemented using a dynamic tree data structure, this approach produces one of the best polynomial time algorithms for solving the minimum cost flow problem.

In this discussion we assume that the reader is familiar with the capacity scaling algorithm and the cost scaling algorithm that we examined in the preceding two sections. To solve the capacitated minimum cost flow problem, we first transform it into an uncapacitated transportation problem using the transformation described in Section 2.4. We assume that every arc in the minimum cost flow problem is capacitated. Consequently, the transformed network will be a bipartite network $G = (N_1 \cup N_2, A)$ with N_1 and N_2 as the sets of supply and demand nodes. Moreover, $| N_1 | = n$ and $| N_2 | = m$.

The double scaling algorithm is the same as the cost scaling algorithm described in the preceding section except that it uses a more efficient version of the improve-approximation procedure. The improve-approximation procedure in the preceding section relied on a "pseudoflow-push" method to push flow out of active nodes. A natural alternative would be to try an augmenting path based method. This approach would send flow from a node with excess to a node with deficit over an *admissible path* (i.e., a path in which each arc is admissible). A straightforward implementation of this approach would require $O(nm)$ augmentations since each augmentation would saturate at least one arc and, by Lemma 10.5, the algorithm requires $O(nm)$ arc saturations. Since each augmentation requires $O(n)$ time, this approach does not appear to improve the $O(n^2m)$ bound of the generic improve-approximation procedure.

We can, however, use ideas from the capacity scaling algorithm to reduce the number of augmentations to $O(m \log U)$ by ensuring that each augmentation carries *sufficiently large* flow. The resulting algorithm performs cost scaling in an "outer loop" to obtain ϵ-optimal flows for successively smaller values of ϵ. Within each cost scaling phase, we start with a pseudoflow and perform a number of capacity scaling phases, called Δ-scaling phases, for successively smaller values of Δ. In the Δ-scaling phase, the algorithm identifies admissible paths from a node with an excess of at least Δ to a node with a deficit and augments Δ units of flow over these paths. When all node excesses are less than Δ, we reduce Δ by a factor of 2 and initiate a new Δ-scaling phase. At the end of the 1-scaling phase, we obtain a flow.

The algorithmic description of the double scaling algorithm is same as that of the cost scaling algorithm except that we replace the improve-approximation procedure by the procedure given in Figure 10.9.

The capacity scaling within the *improve-approximation* procedure is somewhat different from the capacity scaling algorithm described in Section 10.2. The new algorithm differs from the one we considered previously in the following respects:

```
procedure improve-approximation(ε, x, π);
begin
    set x := 0 and compute node imbalances;
    π(j) := π(j) + ε, for all j ∈ N2;
    Δ := 2^⌊log U⌋;
    while the network contains an active node do
    begin
        S(Δ) := {i ∈ N1 ∪ N2 : e(i) ≥ Δ};
        while S(Δ) ≠ ∅ do
        begin {Δ-scaling phase}
            select a node k from S(Δ);
            determine an admissible path P from node k to some node l with e(l) < 0;
            augment Δ units of flow on path P and update x and S(Δ);
        end;
        Δ := Δ/2;
    end;
end;
```

Figure 10.9 Improve-approximation procedure in the double scaling algorithm.

(1) the augmentation terminates at a node l with $e(l) < 0$ but whose deficit may not be as large as Δ; (2) each residual capacity is an integral multiple of Δ because each arc flow is an integral multiple of Δ and each arc capacity is ∞; and (3) the algorithm does not change flow on some arcs at the beginning of the Δ-scaling phase to ensure that the solution satisfies the optimality conditions. We point out that the algorithm feature (3) is a consequence of feature (2) because each r_{ij} is a multiple of Δ, so $G(x, \Delta) \equiv G(x)$.

The double scaling algorithm improves on the capacity scaling algorithm by identifying an admissible path in only $O(n)$ time, on average, rather than the time $O(S(n, m, nC))$ required to identify an augmentation path in the capacity scaling algorithm. The savings in identifying augmenting paths more than offsets the extra requirement of performing $O(\log(nC))$ cost scaling phases in the double scaling algorithm.

We next describe a method for identifying admissible paths efficiently. The algorithm identifies an admissible path by starting at node k and gradually building up the path. It maintains a *partial admissible path* P, which is initially null, and keeps enlarging it until it includes a node with deficit. We maintain the partial admissible path P using predecessor indices [i.e., if $(u, v) \in P$ then $\text{pred}(v) = u$]. At any point in the algorithm, we perform one of the following two steps, whichever is applicable, from the tip of P (say, node i):

advance(i). If the residual network contains an admissible arc (i, j), add (i, j) to P and set $\text{pred}(j) := i$. If $e(j) < 0$, stop.

retreat(i). If the residual network does not contain an admissible arc (i, j), update $\pi(i)$ to $\pi(i) + \epsilon/2$. If $i \neq k$, remove the arc $(\text{pred}(i), i)$ from P so that $\text{pred}(i)$ becomes its new tip.

The retreat step relabels (increases the potential of) node i for the purpose of creating new admissible arcs emanating from this node. However, increasing the potential of node i increases the reduced costs of all the incoming arcs at the node

i by $\epsilon/2$. Consequently, the arc (pred(i), i) becomes inadmissible, so we delete this arc from P (provided that P is nonempty).

We illustrate the method for identifying admissible paths on the example shown in Figure 10.10. Let $\epsilon = 4$ and $\Delta = 4$. Since node 1 is the only node with an excess of at least 4, we begin to develop the admissible path starting from this node. We perform the step advance(1) and add the arc (1, 2) to P. Next, we perform the step advance(2) and add the arc (2, 4) to P. Now node 4 has no admissible arc. So we perform a retreat step. We increase the potential of node 4 by $\epsilon/2 = 2$ units, thus changing the reduced cost of arc (2, 4) to 1; so we eliminate this arc from P. In the next two steps, the algorithm performs the steps advance(2) and advance(5), adding arcs (2, 5) and (5, 6) to P. Since the path now contains node 6, which is a node with a deficit, the method terminates. It has found the admissible path 1–2–5–6.

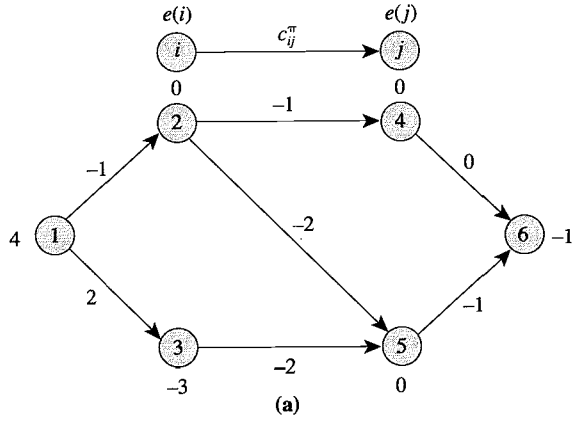


Figure 10.10 Residual network.

It is easy to show that the double scaling algorithm correctly solves the minimum cost flow problem. At the beginning of the improve-approximation procedure, we set $x = 0$ and the corresponding residual network is the same as the original network. The ϵ-optimality of the solution at the end of the previous scaling phase implies that $c_{ij}^{\pi} \geq -\epsilon$ for all arcs $(i, j) \in A$. Therefore, by adding ϵ to $\pi(j)$ for each $j \in N_2$, we obtain an $\frac{1}{2}\epsilon$-optimal pseudoflow (in fact, it is a 0-optimal pseudoflow). Like the improve-approximation procedure described in the preceding section, the algorithm always augments flow on admissible arcs and relabels a node when it has no outgoing admissible arc. Consequently, the algorithm preserves $\frac{1}{2}\epsilon$-optimality of the pseudoflow and at termination yields a $\frac{1}{2}\epsilon$-optimal flow.

We next consider the complexity of the improve-approximation procedure. Each execution of the procedure performs $(1 + \lfloor \log U \rfloor)$ capacity scaling phases. At the end of the 2Δ-scaling phase, $S(2\Delta) = \phi$. Therefore, at the beginning of the Δ-scaling phase, $\Delta \leq e(i) < 2\Delta$ for each node $i \in S(\Delta)$. During the Δ-scaling phase, the algorithm augments Δ units of flow from a node k in $S(\Delta)$ to a node l with $e(l) < 0$. The augmentation reduces the excess of node k to a value less than Δ and ensures that the imbalance at node l is strictly less than Δ. Consequently, each augmentation deletes a node from $S(\Delta)$ and after at most $| N_1 | + | N_2 | = O(m)$ augmentations, $S(\Delta)$ becomes empty and the algorithm begins a new capacity scaling phase. The algorithm thus performs a total of $O(m \log U)$ augmentations.

We next focus on the time needed to identify admissible paths. We first count the number of advance steps. Each advance step adds an arc to the partial admissible path, and each retreat step deletes an arc from the partial admissible path. Thus we can distinguish between two types of advance steps: (1) those that add arcs to an admissible path on which the algorithm later performs an augmentation, and (2) those that are later canceled by a retreat step. Since the set of admissible arcs is acyclic (by Lemma 10.6), after at most $2n$ advance steps of the first type, the algorithm will discover an admissible path and will perform an augmentation (because the longest path in the network has $2n$ nodes). Since the algorithm performs a total of $O(m \log U)$ augmentations, the number of advance steps of the first type is at most $O(nm \log U)$. The algorithm performs $O(nm)$ advance steps of the second type because each retreat step increases a node potential, and by Lemma 10.4, node potentials increase $O(n(n + m)) = O(nm)$ times. Therefore, the total number of advance steps is $O(nm \log U)$.

The amount of time needed to relabel nodes in N_1 is $O(n \sum_{i \in N} | A(i) |) = O(nm)$. The time needed to relabel nodes in N_2 is also $O(nm)$ since $| N_2 | = m$ and the degree of each node in N_2 is constant (i.e., it is 2). The same arguments show that the algorithm requires $O(nm)$ time to identify admissible arcs. We have, therefore, established the following result.

Theorem 10.10. *The double scaling algorithm solves the minimum cost flow problem in $O(nm \log U \log(nC))$ time.* ◆

One nice feature of the double scaling algorithm is that it achieves an excellent worst-case running time for solving the minimum cost flow problem and yet is fairly simple, both conceptually and computationally.

10.5 MINIMUM MEAN CYCLE-CANCELING ALGORITHM

The three minimum cost flow algorithms we have discussed in this chapter—the capacity scaling algorithm, the cost scaling algorithm, and the double scaling algorithm—are weakly polynomial-time algorithms because their running times depend on $\log U$ and/or $\log C$. Although these algorithms are capable of solving any problem with integer or rational data, they are not applicable to problems with irrational data. In contrast, the running times of strongly polynomial-time algorithms depend only on n and m; consequently, these algorithms are capable of solving problems with irrational data, assuming that a computer can perform additions and subtractions on irrational data. In this and the next two sections, we discuss several strongly polynomial time algorithms for solving any class of minimum cost flow problems, including those with irrational data.

The algorithm discussed in this section is a special case of the cycle-canceling algorithm that we discussed in Section 9.6. Because this algorithm iteratively cancels cycles (i.e., augments flows along cycles) with the minimum mean cost in the residential network, it is known as the (*minimum*) *mean cycle-canceling algorithm*. Recall from Section 5.7 that the *mean cost* of a directed cycle W is $(\sum_{(i,j) \in W} c_{ij})/| W |$, and that the *minimum mean cycle* is a cycle with the smallest mean cost in

the network. In Section 5.7 we showed how to use dynamic programming algorithm to find the minimum mean cycle in $O(nm)$ time.

The minimum mean cycle-canceling algorithm starts with a feasible flow x in the network. At every iteration, the algorithm identifies a minimum mean cycle W in $G(x)$. If the mean cost of the cycle W is negative, the algorithm augments the maximum possible flow along W, updates $G(x)$, and repeats this process. If the mean cost of W is nonnegative, $G(x)$ contains no negative cycle and x is a minimum cost flow, so the algorithm terminates. This algorithm is surprisingly simple to state; even more surprisingly, the algorithm runs in strongly polynomial time.

To establish the worst-case complexity of the minimum mean cycle-canceling algorithm, we recall a few facts. In our subsequent discussion, we often use Property 9.2(b), which states that for any set of node potentials π and any directed cycle W, the sum of the costs of the arcs in W equals the sum of the reduced costs of the arcs in W. We will also use the following property concerning sequences of real numbers, which is a variant of the geometric improvement argument (see Section 3.3).

Property 10.11. *Let α be a positive integer and let $y_1, y_2, y_3, \ldots$ be a sequence of real numbers satisfying the condition $y_{k+1} \leq (1 - 1/\alpha)y_k$ for every k. Then for every value of k, $y_{k+\alpha} \leq y_k/2$.*

Proof. We first rewrite the expression $y_{k+1} \leq (1 - 1/\alpha)y_k$ as $y_k \geq y_{k+1} + y_{k+1}/(\alpha - 1)$. We now use this last expression repeatedly to replace the first term on the right-hand side, giving

$$\begin{aligned} y_k &\geq y_{k+1} + y_{k+1}/(\alpha - 1) \geq y_{k+2} + y_{k+2}/(\alpha + 1) + y_{k+1}/(\alpha - 1) \\ &\geq y_{k+2} + 2y_{k+2}/(\alpha - 1) \geq y_{k+3} + 3y_{k+3}/(\alpha - 1) \\ &\vdots \\ &\geq y_{k+\alpha} + \alpha y_{k+\alpha}/(\alpha - 1) \geq 2y_{k+\alpha}, \end{aligned}$$

which is the assertion of the property. ◆

We divide the worst-case analysis of the minimum mean cycle algorithm into two parts: First, we show that the algorithm is weakly polynomial-time; then we establish its strong polynomiality. Although the description of the algorithm does not use scaling techniques, the worst-cast analysis borrows ideas from the cost scaling algorithm that we discussed in Section 10.3. In particular, the notion of ϵ-optimality discussed in that section plays a crucial role in its analysis. We will show that the flows maintained by the minimum mean cycle-canceling algorithm are ϵ-optimal flows satisfying the conditions that (1) between any two consecutive iterations the value of ϵ either stays the same or decreases; (2) occasionally, the value of ϵ strictly decreases; and (3) eventually, $\epsilon < 1/n$ and the algorithm terminates (see Lemma 10.2). As we observed in Section 10.3, the cost scaling algorithm's explicit strategy is to reduce ϵ from iteration to iteration. Although the minimum mean cycle-canceling algorithm also reduces the value of ϵ (although periodically, rather than at every iteration), the reduction is very much an implicit by-product of the algorithm.

We first establish a connection between the ϵ-optimality of a flow x and the

mean cost of a minimum mean cycle in $G(x)$. Recall that a flow x is ϵ-optimal if for some set of node potentials, the reduced cost of every arc is at least $-\epsilon$. Notice that any flow x will be ϵ-optimal for many values of ϵ, because a flow that is ϵ-optimal is also ϵ'-optimal for all $\epsilon' \geq \epsilon$. For any particular set of node potentials π, we let $\epsilon^{\pi}(x)$ be the negative of the minimum value of any reduced cost [i.e., $\epsilon^{\pi}(x) = -\min[c_{ij}^{\pi} : (i, j) \text{ in } G(x)]$. Thus $c_{ij}^{\pi} \geq -\epsilon^{\pi}(x)$ and $c_{ij}^{\pi} = -\epsilon^{\pi}(x)$ for some arc (i, j). Thus x is ϵ-optimal for $\epsilon = \epsilon^{\pi}(x)$. Potentially, we could find a smaller value of ϵ by using other values of the node potentials. With this thought in mind, we let $\epsilon(x) = \min_{\pi} \epsilon^{\pi}(x)$. Note that $\epsilon(x)$ is the smallest value of ϵ for which the flow x is ϵ-optimal. As additional notation, we let $\mu(x)$ denote the mean cost of the minimum mean cycle in $G(x)$.

Note that since x is $\epsilon(x)$-optimal, conditions (10.2) imply that $\sum_{(i,j)\in W} c_{ij} = \sum_{(i,j)\in W} c_{ij}^{\pi} \geq -\epsilon^{\pi}(x) \mid W \mid$. Choosing W as the minimum mean cycle and dividing this expression by $\mid W \mid$, we see that $\mu(x) \geq -\epsilon(x)$. As we have seen, this inequality is a simple consequence of the definitions of ϵ-optimality and of the minimum mean cycle cost; it uses the fact that if we can bound the reduced cost of every arc around a cycle, this same bound applies to the average cost around the cycle. Perhaps surprisingly, however, we can obtain a converse result: that is, we can always find a set of node potentials so that every arc around the minimum mean cycle has the same reduced cost and that this cost equals $-\epsilon(x)$. Our next two results establish this property.

Lemma 10.12. *Let x be a nonoptimal flow. Then $\epsilon(x) = -\mu(x)$.*

Proof. Since our observation in the preceding paragraph shows that $\epsilon(x) \geq -\mu(x)$, we only need to show that $\epsilon(x) \leq -\mu(x)$.

Let W be a minimum mean cycle in the residual network $G(x)$, and let $\mu(x)$ be the mean cost of this cycle. Suppose that we replace each arc cost c_{ij} by $c'_{ij} = c_{ij} - \mu(x)$. This transformation reduces the mean cost of every directed cycle in $G(x)$ by $\mu(x)$ units. Consequently, the minimum mean cost of the cycle W becomes zero, which implies that the residual network contains no negative cost cycle. Let $d'(\cdot)$ denote the shortest path distances in $G(x)$ from a specified node s to all other nodes with c'_{ij} as the arc lengths. The shortest path optimality conditions imply that

$$d'(j) \leq d'(i) + c'_{ij} = d'(i) + c_{ij} - \mu(x) \qquad \text{for each arc } (i, j) \text{ in } G(x). \tag{10.7}$$

If we let $\pi(j) = d'(j)$, then (10.7) becomes

$$c_{ij}^{\pi} \geq \mu(x) \qquad \text{for each arc } (i, j) \text{ in } G(x), \tag{10.8}$$

which implies that x is $(-\mu(x))$-optimal. Therefore, $\epsilon(x) \leq -\mu(x)$, completing the proof of the lemma. ◆

Lemma 10.13. *Let x be any nonoptimal flow. Then for some set of node potentials π, $c_{ij}^{\pi} = \mu(x) = -\epsilon(x)$ for every arc (i, j) in the minimum mean cycle W of $G(x)$.*

Proof. Let π be defined as in the proof of the preceding lemma; with these set of node potentials, the reduced costs satisfy (10.8). The cost of the cycle W equals $\sum_{(i,j)\in W} c_{ij}$, which also equals its reduced cost $\sum_{(i,j)\in W} c_{ij}^{\pi}$. Con-

sequently, $\sum_{(i,j)\in W} c_{ij}^{\pi} = \mu(x) \mid W \mid$. This equation and (10.8) imply that $c_{ij}^{\pi} = \mu(x)$ for each arc (i, j) in W. Lemma 10.12 establishes that $c_{ij}^{\pi} = -\epsilon(x)$ for every arc in W. ◆

We next show that during the execution of the minimum mean cycle-canceling algorithm, $\epsilon(x)$ never increases; moreover, within m consecutive iterations $\epsilon(x)$ decreases by a factor of at least $(1 - 1/n)$.

Lemma 10.14. *For a nonoptimal flow x, if we cancel a minimum mean cycle in $G(x)$, $\epsilon(x)$ cannot increase [alternatively, $\mu(x)$ cannot decrease].*

Proof. Let W denote the minimum mean cycle in $G(x)$. Lemma 10.13 implies that for some set of node potentials π, $c_{ij}^{\pi} = -\epsilon(x)$ for each arc $(i, j) \in W$. Let x' denote the flow obtained after we have canceled the cycle W. This flow augmentation deletes some arcs in W from the residual network and adds some other arcs, which are reversals of the arcs in W. Consider any arc (i, j) in $G(x')$. If (i, j) is in $G(x)$, then, by hypothesis, $c_{ij}^{\pi} \geq -\epsilon(x)$. If (i, j) is not in $G(x)$, then (i, j) is a reversal of some arc (j, i) in $G(x)$ for which $c_{ji}^{\pi} = -\epsilon(x)$. Therefore, $c_{ij}^{\pi} = -c_{ji}^{\pi} = \epsilon(x) > 0$. In either case, $c_{ij}^{\pi} \geq -\epsilon(x)$ for each arc (i, j) in $G(x')$. Consequently, the minimum mean cost of any cycle in $G(x')$ will be at least $-\epsilon(x)$, since the mean cost around a cycle, which equals the mean reduced cost, must be at least as large as the minimum value of the reduced costs. Therefore, in light of Lemma 10.12, as asserted, $\epsilon(x') = \mu(x') \geq -\epsilon(x) = \mu(x)$. ◆

Lemma 10.15. *After a sequence of m minimum mean cycle cancelations starting with a flow x, the value of the optimality parameter $\epsilon(x)$ deceases to a value at most $(1 - 1/n)\,\epsilon(x)$ [i.e., to at most $(1 - 1/n)$ times its original value].*

Proof. Let π denote a set of node potentials satisfying the conditions $c_{ij}^{\pi} \geq -\epsilon(x)$ for each arc (i, j) in $G(x)$. For convenience, we designate those arcs in $G(x)$ with (strictly) negative reduced costs as *negative arcs* (with respect to the reduced costs). We now classify the subsequent cycle cancelations into two types: (1) all the arcs in the canceled cycle are negative (a type 1 cancelation), and (2) at least one arc in the canceled cycle has a nonnegative reduced cost (a type 2 cancelation). We claim that the algorithm will perform at most m type 1 cancelations before it either terminates or performs a type 2 cancelation. This claim follows from the observations that each type 1 cancelation deletes at least one negative arc from the (current) residual network and all the arcs that the cancelation adds to the residual network have positive reduced cost with respect to π (as shown in the proof of Lemma 10.14). Consequently, if within m iterations, the algorithm performs no type 2 cancelations, all the arcs in the residual network will have nonnegative reduced costs with respect to π and the algorithm will terminate with an optimal flow.

Now consider the first time the algorithm performs a type 2 cancelation. Suppose that the algorithm cancels the cycle W, which contains at least one arc with a nonnegative reduced cost; let x' and x'' denote the flows just before and after the cancelation. Then $c_{ij}^{\pi} \geq -\epsilon(x')$ for each arc $(i, j) \in W$ and $c_{kl}^{\pi} \geq 0$ for some arc $(k, l) \in W$. As a result, since $c(W) = \sum_{(i,j)\in W} c_{ij}^{\pi}$, the cost $c(W)$ of W with respect to the flow x' satisfies the condition $c(W) \geq [(\mid W \mid - 1)(-\epsilon(x'))]$. By Lemma 10.14,

the cancelation cannot increase the minimum mean cost and, therefore, $\mu(x'') \geq \mu(x')$. But since $\mu(x')$ is the mean cost of W with respect to x', $\mu(x'') \geq \mu(x') \geq (1 - 1/|W|)(-\epsilon(x')) \geq (1 - 1/n)(-\epsilon(x'))$. This inequality implies that $-\mu(x'') \leq (1 - 1/n)\epsilon(x')$. Using the fact that $\mu(x'') = -\epsilon(x'')$, we see that $\epsilon(x'') \leq (1 - 1/n)\epsilon(x')$. This result establishes the lemma. ◆

As indicated by the next theorem, the preceding two lemmas imply that the minimum mean cycle-canceling algorithm performs a polynomial number of iterations.

Theorem 10.16. *If all arc costs are integer, the minimum mean cycle-canceling algorithm performs $O(nm \log(nC))$ iterations and runs in $O(n^2m^2 \log(nC))$ time.*

Proof. Let x denote the flow at any point during the execution of the algorithm. Initially, $\epsilon(x) \leq C$ because every flow is C-optimal (see Lemma 10.2). In every m consecutive iterations, the algorithm decreases $\epsilon(x)$ by a factor of $(1 - 1/n)$. When $\epsilon(x) < 1/n$, the algorithm terminates with an optimal flow (see Lemma 10.2). Therefore, the algorithm needs to decrease $\epsilon(x)$ by a factor of nC over all iterations. By Lemma 10.15, the mean cost of a cycle becomes smaller by a factor of at least $(1 - 1/n)$ in every m iterations. Property 10.11 implies that the minimum mean cycle cost decreases by a factor of 2 every nm iterations, so that within $nm \log(nC)$ iterations, the minimum mean cycle cost decreases from C to $1/n$. At this point the algorithm terminates with an optimal flow. This conclusion establishes the first part of the theorem. Since the bottleneck operation in each iteration is identifying a minimum mean cycle, which requires $O(nm)$ time (see Section 5.7), we also have established the second part of the theorem.

Having proved that the minimum mean cycle-canceling algorithm runs in polynomial time, we next obtain a strongly polynomial bound on the number of iterations the algorithm performs. Our analysis rests upon the following rather useful result: If the absolute value of the reduced cost of an arc (k, l) is "significantly greater than" the current value of the parameter $\epsilon(x)$, the flow on the arc (k, l) in any optimal solution is the same as the current flow on this arc. In other words, the flow on the arc (k, l) becomes "fixed." As we will show, in every $O(nm \log n)$ iterations, the algorithm will fix at least one additional arc at its lower bound or at its upper bound. As a result, within $O(nm^2 \log n)$ iterations, the algorithm will have fixed all the arcs and will terminate with an optimal flow.

We define an arc to be *ϵ-fixed* if the flow on this arc is the same for all ϵ'-optimal flows whenever $\epsilon' \leq \epsilon$. Since the value of $\epsilon(x)$ of the $\epsilon(x)$-optimal flows, that the minimum mean cycle-canceling algorithm maintains, is nonincreasing, the flow on an $\epsilon(x)$-fixed arc will not change during the execution of the algorithm and will be the same in every optimal flow. We next establish a condition that will permit us to fix an arc.

Lemma 10.17. *Suppose that x is an $\epsilon(x)$-optimal flow with respect to the potentials π, and suppose that for some arc $(k, l) \in A$, $|c_{kl}^{\pi}| \geq 2n\epsilon(x)$. Then arc (k, l) is an $\epsilon(x)$-fixed arc.*

Proof. Let $\epsilon = \epsilon(x)$. We first prove the lemma when $c_{kl}^{\pi} \geq 2n\epsilon$. The ϵ-optimality condition (10.1a) implies that $x_{kl} = 0$. Suppose that some $\epsilon(x')$-optimal flow x', with $\epsilon(x') \leq \epsilon(x)$, satisfies the condition that $x'_{kl} > 0$. The flow decomposition theorem (i.e., Theorem 3.5) implies that we can express x' as x plus the flow along at most m augmenting cycles in $G(x)$. Since $x_{kl} = 0$ and $x'_{kl} > 0$, one of these cycles, say W, must contain the arc (k, l) as a forward arc. Since each arc $(i, j) \in W$ is in the residual network $G(x)$, and so satisfies the condition $c_{ij}^{\pi} \geq -\epsilon$, the reduced cost (or, cost) of the cycle W is at least $c_{kl}^{\pi} - \epsilon(|W| - 1) \geq 2n\epsilon - \epsilon(n - 1) > n\epsilon$.

Now consider the cycle W^r obtained by reversing the arcs in W. The cycle W^r must be a directed cycle in the residual network $G(x')$ (see Exercise 10.6). The cost of the cycle W^r is the negative of the cost of the cycle W and so must be less than $-n\epsilon \leq -n\epsilon(x')$. Therefore, the mean cost of W^r is less than $-\epsilon(x')$. Lemma 10.12 implies that x' is not $\epsilon(x')$-optimal, which is a contradiction.

We next consider the case when $c_{kl}^{\pi} \leq -2n\epsilon$. In this case the ϵ-optimality condition (10.1c) implies that $x_{kl} = u_{kl}$. Using an analysis similar to the one used in the preceding case, we can show that no ϵ-optimal flow x' can satisfy the condition $x'_{kl} < u_{kl}$. ◆

We are now in a position to obtain a strongly polynomial bound on the number of iterations performed by the minimum mean cycle-canceling algorithm.

Theorem 10.18. *For arbitrary real-valued arc costs, the minimum mean cycle-canceling algorithm performs $O(nm^2 \log n)$ iterations and runs in $O(n^2m^3 \log n)$ time.*

Proof. Let $K = nm(\lceil \log n \rceil + 1)$. We divide the iterations performed by the algorithm into groups of K consecutive iterations. We claim that each group of iterations fixes the flow on an additional arc (k, l) (i.e., the iterations after those in the group do not change the value of x_{kl}). The theorem follows immediately from this claim, since the algorithm can fix at most m arcs, and each iteration requires $O(nm)$ time.

Consider any group of iterations. Let x be the flow before the first iteration of the group and let x' be the flow after the last iteration of the group. Let $\epsilon = \epsilon(x)$, $\epsilon' = \epsilon(x')$, and let π' be the node potentials for which x' satisfies the ϵ'-optimality conditions. Since every nm iterations reduce ϵ by a factor of at least 2, the nm $(\lceil \log n \rceil + 1)$ iterations between x and x' reduce ϵ by a factor of at least $2^{\lceil \log n \rceil + 1}$. Therefore, $\epsilon' \leq (\epsilon/2^{\lceil \log n \rceil + 1}) \leq \epsilon/2n$. Alternatively, $-\epsilon \leq -2n\epsilon'$.

Let W be the cycle canceled when the flow has value x. Lemma 10.12 and the fact that the sum of the costs and reduced costs around every cycle are the same, imply that for any values of the node potentials, the average reduced cost around the cycle W equals $\mu(x) = -\epsilon$. Therefore, with respect to the potentials π', at least one arc (k, l) in W must have a reduced cost as small as $-\epsilon$, so $c_{kl}^{\pi'} = -\epsilon \leq -2n\epsilon'$ for some arc (k, l) in W. By Lemma 10.17, the flow on arc (k, l) will not change in any subsequent iteration. Next notice that in the first iteration in the group, the algorithm changed the value of x_{kl}. Thus each group fixes the flow on at least one additional arc, completing the proof of the theorem. ◆

We might conclude this section with a few observations. First, note that we need not formally compute the value of $\epsilon(x)$ at each iteration, nor do we need to identify the ϵ-fixed arcs at any stage in the algorithm. Indeed, we can use any method to find the minimum mean cost cycle at each step; in principle, we need not maintain or ever compute any reduced costs. As we noted earlier in this section, the minimum mean cycle-canceling algorithm implicitly reduces $\epsilon(x)$ and fixes some arcs as it proceeds—we need not keep track of the algorithm's progress concerning these features.

We also might note that the ideas presented in this section would also permit us to develop a strongly polynomial-time version of the cost scaling algorithm that we discussed in Section 10.3. In Exercise 10.12 we consider this modification of the cost scaling algorithm and analyze its running time.

10.6 REPEATED CAPACITY SCALING ALGORITHM

The minimum cost flow problem described in Section 10.5 uses the idea that whenever the reduced cost of an arc is *sufficiently large*, we can "fix" the flow on the arc. By incorporating a similar idea in the capacity scaling algorithm, we can develop another strongly polynomial time algorithm. As we will see, when the flow on an arc (i, j) is *sufficiently large*, the potentials of nodes i and j become "fixed" with respect to each other. In this section we discuss the details of this algorithm, which we call the *repeated capacity scaling algorithm*.

The repeated capacity scaling algorithm to be discussed in this section is different from all the other minimum cost flow algorithms discussed in this book. All of the other algorithms solve the primal minimum cost flow problem (9.1) and obtain an optimal flow; the repeated capacity scaling algorithm solves the dual minimum cost flow problem (9.10). This algorithm obtains an optimal set of node potentials for (9.10) and then uses it to determine an optimal flow.

The repeated capacity scaling algorithm is a modified version of the capacity scaling algorithm discussed in Section 10.2. For simplicity, we describe the algorithm for the uncapacitated minimum cost flow problem; we could solve the capacitated problem by converting it to the uncapacitated problem using the transformation described in Section 2.4. Recall that in the capacity scaling algorithm, each arc flow is an integral multiple of the scale factor Δ. For uncapacitated networks, each residual capacity r_{ij} is also an integral multiple of Δ, because either $r_{ij} = u_{ij} = \infty$, or $r_{ij} = x_{ji} = k\Delta$ for some integer k. This observation implies that the Δ-residual network $G(x, \Delta)$ is the same as the residual network $G(x)$. As a result, the algorithm for the uncapacitated problem does not require the preprocessing (i.e., saturating the arcs violating the optimality conditions) at the beginning of each scaling phase. The following property is an immediate consequence of this result.

Property 10.19. *The capacity scaling algorithm for the uncapacitated minimum cost flow problem satisfies the following properties: (a) the excesses at the nodes are monotonically decreasing; (b) the sum of the excesses at the beginning of the Δ-scaling phase is at most $2n\Delta$; and (c) the algorithm performs at most $2n$ augmentations per scaling phase.*

The repeated capacity scaling algorithm is based on the three simple results stated in the following lemmas.

Lemma 10.20. *Suppose that at the beginning of the Δ-scaling phase, $b(k) > 6n^2\Delta$ for some node $k \in N$. Then some arc (k, l) with $x_{kl} > 4n\Delta$ emanates from node k.*

Proof. Property 10.19 implies that at the beginning of the Δ-scaling phase, the sum of the excesses is at most $2n\Delta$. Therefore, $e(k) \leq 2n\Delta$. Since $b(k) > 6n^2\Delta$ and $e(k) \leq 2n\Delta$, the net outflow of node k [i.e., $b(k) - e(k)$] is strictly greater than $(6n^2\Delta - 2n\Delta)$. Since fewer than n arcs emanate from node k, the flow on at least one of these arcs must be strictly more than $(6n^2\Delta - 2n\Delta)/n \geq (4n^2\Delta)/n = 4n\Delta$, which concludes the lemma. ◆

Lemma 10.21. *If at the beginning of the Δ-scaling phase $x_{kl} > 4n\Delta$, then for some optimal solution $x_{kl} > 0$.*

Proof. Property 10.19 implies that the algorithm performs at most $2n$ augmentations in each scaling phase. The fact that the algorithm augments exactly Δ units of flow in every augmentation in the Δ-scaling phase implies that the total flow change due to all augmentations in the subsequent scaling phases is at most $2n(\Delta + \Delta/2 + \Delta/4 + \cdots + 1) < 4n\Delta$. Consequently, if $x_{kl} > 4n\Delta$ at the beginning of the Δ-scaling phase, then $x_{kl} > 0$ when the algorithm terminates. ◆

Lemma 10.22. *Suppose that $x_{kl} > 0$ in an optimal solution of the minimum cost flow problem. Then with respect to every set of optimal node potentials, the reduced cost of arc (k, l) is zero.*

Proof. Suppose that x satisfies the complementary slackness optimality condition (9.8) with respect to the node potential π. The condition (9.8b) implies that $c^{\pi}_{kl} = 0$. Property 9.8 implies that if x satisfies the complementary slackness optimality condition (9.8b) with respect to some node potential, it satisfies this condition with respect to every optimal node potential. Consequently, the reduced cost of arc (k, l) is zero with respect to every set of optimal node potentials. ◆

We are now in a position to discuss the essential ideas of the repeated capacity scaling algorithm. Let **P** denote the minimum cost flow problem stated in (9.1). The algorithm applies the capacity scaling algorithm stated in Figure 10.1 to the problem **P.** We will show that within $O(\log n)$ scaling phases, $b(k) > 6n^2\Delta$ for some node k and, by Lemma 10.20, some arc (k, l) satisfies the condition $x_{kl} > 4n\Delta$. Lemmas 10.21 and 10.22 imply that for any set of optimal node potentials, the reduced cost of arc (k, l) will be zero. This result allows us to show, as described next, that we can *contract* the nodes k and l into a single node, thereby obtaining a new minimum cost flow problem defined on a network with one fewer node.

Suppose that we are using the capacity scaling algorithm to solve a minimum cost flow problem **P** with arc costs c_{ij} and at some stage we realize that for an arc (k, l), $x_{kl} > 4n\Delta$. Let π denote the node potentials at this point. The optimality condition (9.8b) implies that

$$c_{kl} - \pi(k) + \pi(l) = 0. \tag{10.9}$$

Now consider the same minimum cost flow problem, but with the cost of each arc (i, j) equal to $c'_{ij} = c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j)$. Let $\mathbf{P}'$ denote the modified minimum cost flow problem. Condition (10.9) implies that

$$c'_{kl} = 0. \tag{10.10}$$

We next observe that the problems $\mathbf{P}$ and $\mathbf{P}'$ have the same optimal solutions (see Property 2.4 in Section 2.4). Since $x_{kl} > 4n\Delta$, Lemmas 10.21 and 10.22 imply that in problem $\mathbf{P}'$ the reduced cost of arc (k, l) will be zero. If π' denotes an optimal set of node potentials for $\mathbf{P}'$, then

$$c'_{kl} - \pi'(k) + \pi'(l) = 0. \tag{10.11}$$

Substituting (10.10) in (10.11) implies that $\pi'(k) = \pi'(l)$.

The preceding discussion shows that if $x_{kl} > 4n\Delta$ for some arc (k, l), we can "fix" one node potential with respect to the other. The discussion also shows that if we solve the problem $\mathbf{P}'$ with the additional constraint that the potentials of nodes k and l are same, this constraint will not eliminate the optimal solution of $\mathbf{P}'$. But how can we solve a minimum cost flow problem when two node potentials must be the same?

Consider the dual minimum cost flow problem stated in (9.10). In this problem we replace both $\pi(k)$ and $\pi(l)$ by $\pi(p)$. This substitution gives us a linear programming problem with one less dual variable (or, node potential). The reader can easily verify that the resulting problem is a dual minimum cost flow problem on the network with nodes k and l contracted into a single node p. The contraction operation consists of (1) letting $b(p) = b(k) + b(l)$, (2) replacing each arc (i, k) or (i, l) by the arc (i, p), (3) replacing each arc (k, i) or (l, i) by the arc (p, i), and (4) letting the cost of an arc in the contracted network equal that of the arc it replaces. We point out that the contraction might produce multiarcs (i.e., more than one arc with the same tail and head nodes). The purpose of contraction operations should be clear; since each contraction operation reduces the size of the network by one node, we can apply at most n of these operations.

We can now describe the repeated capacity scaling algorithm. We first compute $U = \max\{b(i) : i \in N \text{ and } b(i) > 0\}$ and initialize $\Delta = 2^{\lfloor \log U \rfloor}$. Let node k be a node with $b(k) = U$. We then apply the capacity scaling algorithm as described in Figure 10.1. Each scaling phase of the capacity scaling algorithm decreases Δ by a factor of 2; therefore, since the initial value of Δ is $b(k)$, after at most $q = \log (6n^2) = O(\log n)$ phases, $\Delta = b(k)/2^q \leq b(k)/6n^2$. The algorithm might obtain a feasible flow before $\Delta \leq b(k)/6n^2$ (in which case it terminates); if not, then by Lemma 10.20, some arc (k, l) will satisfy the condition that $x_{kl} > 4n\Delta$. The algorithm then defines a new minimum cost flow problem with nodes k and l contracted into a new node p, and the cost of each arc is the reduced cost of the corresponding arc before the contraction. We solve the new minimum cost flow problem afresh by redefining U as the largest supply in the contracted network and reapplying the capacity scaling algorithm described in Figure 10.1. We repeat these steps until the algorithm terminates. The algorithm terminates in one of the two ways: (1) while applying the capacity scaling algorithm, it obtains a flow; or (2) it contracts the network into a

single node p [with $b(p) = 0$], which is trivially solvable by a zero flow. At this point we expand the contracted nodes and obtain an optimal flow in the expanded network. We show how to expand the contracted nodes a little later. The preceding discussion shows that the algorithm performs $O(n \log n)$ scaling phases, and since each scaling phase solves at most $2n$ shortest path problems, the running time of the algorithm is $O(n^2 \log n\, S(n, m))$. In this expression, $S(n, m)$ is the minimum time required by a strongly polynomial-time algorithm for solving a shortest path problem with nonnegative arc lengths. [Recall from Chapter 4 that $O(m + n \log n)$ is currently the best known such bound.]

We illustrate the repeated capacity scaling algorithm on the example shown in Figure 10.11(a). When applied to this example, the capacity scaling algorithm performs 100 scaling phases with $\Delta = 2^{99}, 2^{98-1}, \ldots, 2^0$. The strongly polynomial version, however, terminates within five phases, as shown next.

Phase 1. In this phase, $\Delta = 2^{99}$, $S(\Delta) = \{1, 2\}$, and $T(\Delta) = \{3, 4\}$. The algorithm augments Δ units of flow along the two paths 1–3 and 2–1–3–4. Figure 10.11(b) shows the solution at the end of this phase.

Phase 2. In this phase, $\Delta = 2^{98}$. The algorithm augments Δ units of flow along the path 1–3.

Phase 3. In this phase, $\Delta = 2^{97}$. The algorithm augments Δ units of flow along the path 1–3.

Phase 4. In this phase, $\Delta = 2^{96}$. The algorithm finds that the flow on the arc (1, 3) is $2^{100} + 2^{99} + 2^{98}$, which is more than $4n\Delta = 2^{100}$. Therefore, the algorithm contracts the nodes 1 and 3 into a new node 5 and obtains the minimum cost flow problem shown in Figure 10.11(c), which it then proceeds to solve.

Phase 5. In this phase, $\Delta = 2^{95}$. The algorithm augments Δ units of flow along the path 2–5–4. The solution is a flow now; consequently, the algorithm terminates. The corresponding flow in the original network is $x_{21} = 2^{99}$, $x_{13} = 2^{100} - 1$, and $x_{34} = 2^{99}$.

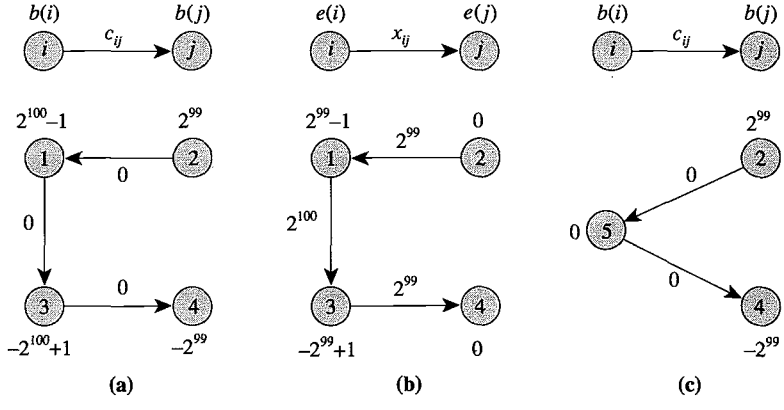


Figure 10.11 Illustrating the repeated capacity scaling algorithm: (a) minimum cost flow problem; (b) solution after the first phase; (c) minimum cost flow problem after contracting the nodes 1 and 3 into a new node 5.

We now explain how we expand the contracted network, and in the process we prove that the algorithm determines an optimal solution of the minimum cost flow problem. The algorithm, in fact, first determines an optimal set of node potentials of the problem, and then by solving a maximum flow problem (as described in Section 9.5) determines an optimal flow. The algorithm obtains an optimal set of node potentials for the original problem by repeated use of the following result.

Property 10.23. *Let* **P** *be a problem with arc costs* c_{ij} *and* **P'** *be the same problem with arc costs* $c_{ij} - \pi(i) + \pi(j)$. *If* π' *is an optimal set of node potentials for problem* **P'**, *then* $\pi + \pi'$ *is an optimal set of node potentials for* **P**.

Proof. This property easily follows from the observation that if a solution x satisfies the reduced cost optimality condition (9.7) with respect to the arc costs $c_{ij} - \pi(i) + \pi(j)$ and node potentials π', the same solution satisfies these conditions with arc costs c_{ij} and node potentials $\pi + \pi'$. ◆

We expand (or uncontract) the nodes in the reverse order in which we contracted them in the strongly polynomial algorithm and obtain optimal node potentials of the successive problems. In earlier stages, between two successive problems, we performed two transformations in the following order: (1) we replaced the arc cost c_{ij} by its reduced cost $c_{ij} - \pi(i) + \pi(j)$, and (2) we contracted two nodes k and l into a single new node p. We undo these transformations in the reverse order. To undo the contracted node p, for case (2) we set the potentials of nodes k and l equal to that of node p, and for case (1) we add π to the existing node potentials. When we have expanded all the contracted nodes, the resulting node potentials are an optimal set of node potentials for the minimum cost flow problem. Then, as described in Section 9.5, we can use these node potentials to obtain an optimal flow by solving a maximum flow problem. The following theorem summarizes the preceding discussion.

Theorem 10.24. *The repeated capacity scaling algorithm solves the uncapacitated minimum cost flow problem in* $O(n^2 \log n\, S(n, m))$ *time.* ◆

Since the best known strongly polynomial-time algorithm for solving the shortest path problem with nonnegative arc lengths runs in $O(m + n \log n)$ time, the best current bound for the uncapacitated minimum cost flow problem is $O(n \log n(m + n \log n))$. We can solve the capacitated minimum cost flow problem by the repeated capacity scaling algorithm by first transforming it to an uncapacitated problem (see Section 2.4). The uncapacitated network will have $n' = n + m$ nodes and $m' = 2m$ arcs. When applied to this network, the repeated capacity scaling algorithm will perform $O(n' \log n') = O(m \log n)$ scaling phases and solve $O(m') = O(m)$ shortest path problems in each scaling phase. Thus the running time of the algorithm is the time needed to solve $O(m^2 \log n)$ shortest path problems. Each shortest path problem in the uncapacitated network requires $O(2m + (m + n) \log (m + n)) = O(m + m \log n)$ time, but using a clever approach for solving the resulting shortest path problem (as discussed in Exercise 4.53) we can obtain a better bound of $O(m + n \log n)$. Consequently, the repeated capacity scaling algorithm requires $O(m^2 \log n(m + n \log n))$ time to solve a capacitated minimum cost flow problem.

10.7 ENHANCED CAPACITY SCALING ALGORITHM

In this section we discuss yet another strongly polynomial-time algorithm for the minimum cost flow problem. This algorithm is a variant of the capacity scaling algorithm that we discussed in Section 10.2 and draws on some ideas from the repeated capacity scaling algorithm discussed in Section 10.6. We refer to this algorithm as the *enhanced capacity scaling algorithm*. This algorithm runs in $O((m \log n)(m + \log n))$ time for the capacitated minimum cost flow problem and is currently the fastest strongly polynomial-time algorithm for solving the minimum cost flow problem. In this section we first show how to solve the enhanced capacity scaling algorithm for the *uncapacitated* minimum cost flow problem; we can solve the capacitated problem by transforming it to an uncapacitated problem (see Section 2.4).

Recall from Section 10.6 that the essential idea in the repeated capacity scaling algorithm is to identify arcs with *sufficiently large* flow. The repeated capacity scaling algorithm identifies such an arc (k, l) within $O(\log n)$ scaling phases, contracts the nodes k and l into a single node, and solves the resulting minimum cost flow problem afresh. For the uncapacitated minimum cost flow problem, this algorithm performs a total of $O(n \log n)$ scaling phases and $O(n^2 \log n)$ shortest path augmentations. The enhanced capacity scaling algorithm adopts a similar approach but it differs in the following two ways: (1) the algorithm does not explicitly perform the contraction operation; and (2) the algorithm does not solve the minimum cost flow problem afresh, but continues from where it left off in its earlier computations. By avoiding contractions, the algorithm achieves ease of coding (because contractions change the network structure and so its computer representation) and maintains a pseudoflow satisfying the dual optimality conditions at every step until the end, at which point it becomes an optimal flow. Moreover, the total number of scaling phases is $O(n \log n)$ and the total number of shortest path augmentations in these scalings phases is also $O(n \log n)$. Consequently, if $S(n, m)$ is the time required to solve a shortest path problem with nonnegative arc lengths, the running time of the enhanced capacity scaling algorithm for uncapacitated problems is $O(n \log n\, S(n, m))$. For capacitated minimum cost flow problems, this time bound becomes $O(m \log n\, S(n, m))$. [By Exercise 4.53 the time bound for the shortest path problem in the transformed network is $O(S(n, m))$ rather than $O(S(n + m, 2m))$ even though the transformed network has $n + m$ nodes and $2m$ arcs.]

The enhanced capacity scaling algorithm proceeds by performing scaling phases for different values of the scale factor Δ. In the Δ-scaling phase, we say that an arc (i, j) has a *sufficiently large* flow if $x_{ij} \geq 8n\Delta$. [We later show that if $x_{ij} \geq 8n\Delta$, then arc (i, j) will have positive flow during the entire execution of the algorithm.] We refer to an arc with sufficiently large flow as an *abundant arc*; otherwise, we call it a *nonabundant arc*. We refer to the subgraph consisting of the node set N and abundant arcs as the *abundant subgraph*. The abundant subgraph typically contains several components, which we call *abundant components*. If the network contains no abundant arc, the abundant subgraph contains n components, each consisting of a singleton node. For simplicity, we will designate an abundant component by the set S of nodes it spans. We let $b(S) = \sum_{i \in S} b(i)$ and $e(S) = \sum_{i \in S} e(i)$.

We designate an (arbitrary) node in each abundant component as its *root* and refer to all the other nodes as *nonroot nodes*. By convention we assume that the

minimum index node in an abundant component is its root. For example, if $S = \{3, 5, 9\}$, then node 3 is the root node of the abundant component S. Throughout its execution, the enhanced capacity scaling algorithm satisfies the following properties.

Property 10.25 (Flow Property). *In the Δ-scaling phase, the flow on each nonabundant arc is an integral multiple of Δ; an abundant arc can have any nonnegative flow value.*

Property 10.26 (Imbalance Property). *Each nonroot node has a zero imbalance; a root node can have an excess or a deficit.*

At the beginning of the enhanced capacity scaling algorithm, the network has no abundant arc and the abundant subgraph contains n components, each consisting of a singleton node. As the algorithm proceeds, it identifies abundant arcs and adds them to the abundant subgraph. Suppose that the algorithm adds a new abundant arc (i, j) at some stage. Let S_i and S_j, respectively, denote the abundant components containing the nodes i and j. If $S_i = S_j$ [i.e., the arc (i, j) has both of its endpoints in the same component], this addition does not create any new abundant component; otherwise, the addition creates a new abundant component consisting of the union of S_i and S_j. We refer to this operation as a *merge* operation because it merges the components S_i and S_j into a single abundant component. Notice that since each merge operation reduces the number of abundant components by one, the algorithm can perform at most n merge operations.

Whenever the algorithm merges the components S_i and S_j, we need to ensure that the solution satisfies the imbalance property. Suppose that i_r and j_r denote the root nodes of the components S_i and S_j before the merge operation. Suppose further that $i_r < j_r$. If $e(j_r) = 0$, after the merge operation the abundant subgraph satisfies the imbalance property. However, if $e(j_r)$ is nonzero, we satisfy the imbalance property by sending $e(j_r)$ units of flow from node j_r to node i_r using any path in the merged component. [Notice that if $e(j_r) < 0$, we should view this augmentation as augmenting $| e(j_r) |$ units of flow from node i_r to j_r so we eliminate the imbalance at node j_r.] Observe that this augmentation changes the flow on some abundant arcs by $| e(j_r) |$ units. We refer to this augmentation as an *imbalance-property augmentation*. In Exercise 10.26 we ask the reader to show how to perform merge operations and the subsequent imbalance-property augmentations in $O(m)$ time.

We are now in a position to describe the enhanced capacity scaling algorithm. Figure 10.12 gives an algorithmic description of this algorithm.

The enhanced capacity scaling algorithm performs two types of augmentations. The first type of augmentation enforces the imbalance property when the algorithm identifies new abundant arcs; we have earlier defined these augmentations as the *imbalance-property augmentations*. The second type of augmentation takes place from excess nodes to deficit nodes along shortest paths. We refer to these augmentations as *shortest-path augmentations*.

As we have already mentioned, the enhanced capacity scaling algorithm is a variant of the capacity scaling algorithm. These two algorithms differ in the following respects:

```
algorithm enhanced capacity scaling;
begin
    set x := 0, π := 0, and e := b;
    set Δ := max{|e(i)| : i ∈ N};
    while the residual network G(x) contains a node i with e(i) > 0 do
    begin
        if max{e(i) : i ∈ N} ≤ Δ/(8n) then Δ := max{e(i) : i ∈ N};
        {the Δ-scaling phase begins here}
        for each nonabundant arc (i, j) do
        if x_ij ≥ 8nΔ then designate arc (i, j) as an abundant arc;
        update abundant components and reinstate the imbalance property;
        while the residual network G(x) contains a node k with | e(k) | ≥ (n − 1)Δ/n do
        begin
            select a pair of nodes k and l satisfying the property that (i) either e(k) > (n − 1)Δ/n
              and e(l) < −Δ/n, or (ii) e(k) > Δ/n and e(l) < −(n − 1)Δ/n;
            considering reduced costs as arc lengths, compute shortest path distance d(·) in
              G(x) from node k to all other nodes;
            π(i) := π(i) − d(i) for all i ∈ N;
            augment Δ units of flow along the shortest path in G(x) from node k to node l;
        end;
        {the Δ-scaling phase ends here}
        Δ := Δ/2;
    end;
end;
```

Figure 10.12 Enhanced capacity scaling algorithm.

1. In the capacity scaling algorithm, we set the initial value of $\Delta = 2^{\lfloor \log U \rfloor}$, that is, the largest power of 2 less than or equal to $U = \max\{|\, b(i) \,| : i \in N\}$. In a strongly polynomial algorithm, we cannot take logarithms because we cannot determine $\log U$ in $O(1)$ elementary arithmetic operations. Therefore, in the enhanced capacity scaling algorithm, we set $\Delta = \max\{|\, b(i) \,| : i \in N\}$.
2. The capacity scaling algorithm decreases Δ by a factor of 2 in every scaling phase. In the enhanced capacity scaling algorithm, we also decrease Δ by a factor of 2, but if $\max\{|\, e(i) \,| : i \in N\} \leq \Delta/8n$, then we reset $\Delta = \max\{|\, e(i) \,| : i \in N\}$. Consequently, the enhanced capacity scaling algorithm generally decreases Δ by a factor of 2, but sometimes by a larger factor when imbalances are too small compared to the current scale factor. Without resetting Δ in this way, the capacity scaling algorithm might perform $O(\log U)$ scaling phases, many of which will not perform any augmentations. The resulting algorithm would contain $O(\log U)$ in its running time and would not be strongly polynomial-time.
3. In the capacity scaling algorithm, each arc flow is an integral multiple of Δ. This property is essential for its correctness because it ensures that each positive residual capacity is a multiple of Δ, and consequently, any augmentation can carry Δ units of flow. In the enhanced capacity scaling algorithm, although the flows on nonabundant arcs are integral multiples of Δ, the flows on the abundant arcs can be arbitrary. Since the flows on abundant arcs are sufficiently large, their arbitrary values do not prohibit sending Δ units of flow on them.
4. The capacity scaling algorithm sends Δ units of flow from a node k with $e(k) \geq \Delta$ to a node l with $e(l) \leq -\Delta$. As a result, the excess nodes do not become

deficit nodes, and vice versa. In the enhanced capacity scaling algorithm, augmentations carry Δ units of flow and are (a) either from a node k with $e(k) > (n-1)\Delta/n$ to a node l with $e(l) < -\Delta/n$, (b) or from a node k with $e(k) > \Delta/n$ to a node l with $e(l) < -(n-1)\Delta/n$. Notice that due to these choices, excess nodes might become deficit nodes and deficit nodes might become excess nodes. Although these choices might seem a bit odd when compared to the capacity scaling algorithm, they ensure several nice theoretical properties that we describe in the following discussion.

We establish the correctness of the enhanced capacity scaling algorithm as follows. In the Δ-scaling phase, we refer to a node i as a *large excess node* if $e(i) > (n-1)\Delta/n$ and as a *medium excess node* if $e(i) > \Delta/n$. (Observe that a large excess node is also a medium excess node.) Similarly, we refer to a node i as a *large deficit node* if $e(i) < -(n-1)\Delta/n$ and as a *medium deficit node* if $e(i) < -\Delta/n$. In the Δ-scaling phase, each shortest path augmentation either starts at a large excess node k and ends at a medium deficit node l, or starts at a medium excess node k and ends at a large deficit node l. To establish the correctness of the algorithm, we need to show that whenever (1) the network contains a large excess node k, it must also contain a medium deficit node l, or when (2) the network contains a large deficit node l, it must also contain a medium excess node k. We establish this result in the following lemma.

Lemma 10.27. *If the network contains a large excess node k, it must also contain a medium deficit node l. Similarly, if the network contains a large deficit node l, it must also contain a medium excess node k.*

Proof. We prove the first part of the lemma; the proof of the second part is similar. Note that $\sum_{i \in N} e(i) = 0$ because the total excess of the excess nodes equals the total deficit of the deficit nodes. If $e(k) > (n-1)\Delta/n$ for some excess node k, the total deficit of deficit nodes is also greater than $(n-1)\Delta/n$. Since the network contains at most $(n-1)$ deficit nodes, at least one of these nodes, say node l, must have a deficit greater than Δ/n, or equivalently $e(l) < -\Delta/n$. ◆

In the proofs, we use the following lemma several times.

Lemma 10.28. *At the end of the Δ-scaling phase, $|e(i)| \le (n-1)\Delta/n$ for each node i. At the beginning of the Δ-scaling phase, $|e(i)| \le 2(n-1)\Delta/n$ for each node i.*

Proof. Suppose that during some scaling phase the network contains some large excess node. Then by Lemma 10.27, it also contains some medium deficit node, so the scaling phase would not yet end. Similarly, if the network contains some large deficit node, it would also contain some medium excess node, and the scaling phase would not end. Therefore, at the end of the scaling phase, $|e(i)| \le (n-1)\Delta/n$ for each node i.

If at the next scaling phase the algorithm halves the value of Δ, then $|e(i)| \le 2(n-1)\Delta/n$ for each node i. On the other hand, if the algorithm sets Δ equal to $e_{\max}$, then $|e(i)| \le \Delta$ for each node i. In either case, the lemma is true. ◆

The enhanced capacity scaling algorithm also relies on the fact that in the Δ-scaling phase, we can send Δ units of flow along the shortest path P from node k to node l. To prove this result, we need to show that the residual capacity of every arc in the path P is at least Δ. We establish this property in two parts. First, we show that the flow on each nonabundant arc is a multiple of Δ; this would imply that residual capacities of nonabundant arcs and their reversals in the residual network are multiples of Δ (because all the arcs in A are uncapacitated). We next show that the flow on each abundant arc is always greater than or equal to $4n\Delta$; therefore, we can send Δ units of flow in either direction. These two results would complete the correctness proof of the enhanced capacity scaling algorithm.

Lemma 10.29. *Throughout the execution of the enhanced capacity scaling algorithm, the solution satisfies the flow and imbalance properties (i.e., Properties 10.25 and 10.26).*

Proof. We prove this lemma by performing induction on the number of flow augmentations and changes in the scale factor Δ. We first consider the flow property. Each augmentation sends Δ units of flow and thus preserves the property. The scale factor Δ changes in one of the two following ways: (1) when we replace Δ by $\Delta' = \Delta/2$, or (2) after replacing $\Delta' = \Delta/2$, we reset $\Delta'' = \max\{| e(i) | : i \in N\}$. In case (1), the flows on the nonabundant arcs continue to be multiples of Δ'. In case (2), $\Delta'' = \max\{e(i) : i \in N\} \leq \Delta'/8n$, or $\Delta' \geq 8n\Delta''$. Since each positive arc flow x_{ij} on a nonabundant arc is a multiple of Δ', $x_{ij} \geq \Delta' \geq 8n\Delta''$. Consequently, each positive flow arc becomes an abundant arc (with respect to the new scale factor) and vacuously satisfies the flow property.

We next establish the imbalance property by performing induction on the number of augmentations and the creation of new abundant arcs. Each augmentation carries flow from a nonroot node to another nonroot node and preserves the property. Moreover, each time the algorithm creates a new abundant arc, it might create a nonroot node i with nonzero imbalance; however, it immediately performs an imbalance-property augmentation to reduce its imbalance to zero. The lemma now follows. ◆

Theorem 10.30. *In the Δ-scaling phase, the algorithm changes the flow on any arc by at most $4n\Delta$ units.*

Proof. The flow on an arc changes through either imbalance-property augmentations or shortest path augmentations. We first consider changes caused by imbalance-property augmentations. At the beginning of the Δ-scaling phase, $e(i) \leq 2(n-1)\Delta/n$ for each node i (from Lemma 10.28). Consequently, an imbalance-property augmentation changes the flow on any arc by at most $2(n-1)\Delta/n$. Since the algorithm can perform at most n imbalance-property augmentations at the beginning of a scaling phase, the change in the flow on an arc due to all imbalance-property augmentations is at most $2(n-1)\Delta \leq 2n\Delta$.

Next consider the changes in the flow on an arc caused by shortest path augmentations. At the beginning of the Δ-scaling phase, each root node i satisfies the condition $| e(i) | \leq 2(n-1)\Delta/n$ (by Lemma 10.29). Consider the case when the Δ-scaling phase performs no imbalance-property augmentations. In this case, at most

one shortest path augmentation will begin at a large excess node i, because after this augmentation, the new excess $e'(i)$ satisfies the inequality $e'(i) \leq 2(n - 1)\Delta/n - \Delta = (n - 2)\Delta/n \leq (n - 1)\Delta/n$, and node i is no longer a large excess node. Similarly, at most one shortest path augmentation will end at a large deficit node.

Now suppose that the algorithm does perform some imbalance-property augmentations. In this case the algorithm sends $e(j)$ units of flow from each nonroot node j to the root of its abundant component. The subsequent imbalance-property augmentation from node j to the root node i can increase $|e(i)|$ by at most $2(n - 1)\Delta/n$ units, so node i can be the start or end node of at most two additional shortest path augmentations in the Δ-scaling phase. We "charge" these two augmentations to node j, which becomes a nonroot node and remains a nonroot node in the subsequent scaling phases.

To summarize, we have shown that in the Δ-scaling phase, we can charge each root node at most one shortest path augmentation and each nonroot node at most two shortest path augmentations. Each such augmentation changes the flow on any arc by 0 or Δ units. Consequently, the total flow change on any arc due to all shortest path augmentations is at most $2n\Delta$. We have earlier shown the total flow change due to imbalance-property augmentations is at most $2n\Delta$. These results establish the theorem. ◆

The preceding theorem immediately implies the following result.

Lemma 10.31. *If the algorithm designates an arc (i, j) as an abundant arc in the Δ-scaling phase, then in all subsequent Δ'-scaling phases $x_{ij} \geq 4n\Delta'$.*

Proof. We prove this result by performing induction on the number of scaling phases. Since the algorithm designates arc (i, j) as an abundant arc at the beginning of the Δ-scaling phase, the flow on this arc satisfies the condition $x_{ij} \geq 8n\Delta$. The Lemma 10.31 implies that the flow change on any arc in the Δ-scaling phase is at most $4n\Delta$. Therefore, throughout the Δ-scaling phase and, also, at the end of this scaling phase, the arc (i, j) satisfies the condition $x_{ij} \geq 4n\Delta$. In the next scaling phase, the scale factor $\Delta' \leq \Delta/2$; so at the beginning of the Δ'-scaling phase, $x_{ij} \geq 8n\Delta'$. This conclusion establishes the lemma. ◆

We next consider the worst-case complexity of the enhanced capacity scaling algorithm. We show that the algorithm performs $O(n \log n)$ scaling phases, requiring a total of $O(n \log n)$ shortest path augmentations. These proofs rely on the result, stated in Theorem 10.33, that any abundant component whose root node has a medium excess or a medium deficit merges into a larger abundant component within $O(\log n)$ scaling phases. Theorem 10.33, in turn, depends on the following lemma.

Lemma 10.32. *Let S be the set of nodes spanned by an abundant component, and let $e(S) = \sum_{i \in S} e(i)$ and $b(S) = \sum_{i \in S} b(i)$. Then $b(S) - e(S)$ is an integral multiple of Δ.*

Proof. Summing the mass balance constraints (9.1b) of nodes in S, we see that

$$b(S) - e(S) = \sum_{\{(i,j) \in (S, \bar{S})\}} x_{ij} - \sum_{\{(i,j) \in (\bar{S}, S)\}} x_{ij}. \tag{10.12}$$

In this expression, $(S, \bar{S})$ and $(\bar{S}, S)$ denote the sets of forward and backward arcs in the cut $[S, \bar{S}]$. Since the flow on each arc in the cut is an integral multiple of Δ (by the flow property), $b(S) - e(S)$ is also an integral multiple of Δ. ◆

Theorem 10.33. *Let S be the set of nodes spanned by an abundant component and suppose that at the end of the Δ-scaling phase, $| e(S) | > \Delta/n$. Then within $O(\log n)$ additional scaling phases, the algorithm will merge the abundant component S into a larger abundant component.*

Proof. We first claim that at the end of the Δ-scaling phase, $| b(S) | \geq \Delta/n$. We prove this result by contradiction. Suppose that $| b(S) | < \Delta/n$. Let node i be the root node of the component S. Lemma 10.28 implies that at the end of the Δ-scaling phase, $| e(i) | = | e(S) | \leq (n - 1)\Delta/n$. Therefore, $| b(S) | + | e(S) | < \Delta$, which from Lemma 10.32 is possible only if $| b(S) | = | e(S) |$. This condition, however, contradicts the facts that $| e(S) | > \Delta/n$ and $| b(S) | < \Delta/n$. Therefore, $| b(S) | \geq \Delta/n$ whenever $| e(S) | > \Delta/n$. Consequently, at the end of the Δ-scaling phase, $| b(S) | \geq \Delta/n$.

Since the enhanced capacity scaling algorithm decreases Δ by a factor of at least 2 in each scaling phase, within $\log (9n^2m) \leq \log (9n^4) = O(\log n)$ scaling phases, the scale factor will be $\Delta' \leq \Delta/2^{\log(9n^2m)} = \Delta/(9n^2m)$, or $\Delta/n \geq 9nm\Delta'$. Since $| b(S) | \geq \Delta/n$, $| b(S) | \geq 9nm\Delta'$. We consider the situation when $b(S) > 0$. [The analysis of the situation with $b(S) < 0$ is similar.] Since $e(S) \leq \Delta'(n - 1)/n \leq \Delta'$ (by Lemma 10.28), the flow across the cut $[S, \bar{S}]$ (i.e., the right-hand side of (10.12)) is at least $9nm\Delta' - \Delta' \geq 8nm\Delta'$. This cut contains at most m arcs; at least one of these arcs, say arc (i, j), must have a flow at least $8n\Delta'$. Thus the algorithm will designate the arc (i, j) as an abundant arc and merge the component S into a larger abundant component. ◆

We are now ready to complete the proof of the main result of this section.

Theorem 10.34. *The enhanced capacity scaling algorithm solves the uncapacitated minimum cost flow problem within $O(n \log n)$ scaling phases and performs a total of $O(n \log n)$ shortest path augmentations. If $S(n, m)$ is the time required to solve a shortest path problem with nonnegative arc lengths, the running time of the enhanced capacity scaling algorithm is $O(n \log n\, S(n, m))$.*

Proof. We first show that the algorithm performs $O(n \log n)$ scaling phases. Consider a scaling phase with scale factor equal to Δ. At the end of this scaling phase, we will encounter one of the following two outcomes:

Case 1. For some node i, $| e(i) | > \Delta/16n$. Let node i be the root node of an abundant component S. Clearly, within four scaling phases, either the component S merges into a larger component or $| e(i) | > \Delta/n$. In the latter case, Theorem 10.33 implies that within $O(\log n)$ scaling phases, the component S merges into a larger component.

Case 2. For every node i, $| e(i) | \leq \Delta/16n$. At the beginning of the next scaling phase, the new scale factor $\Delta' = \Delta/2$, so $| e(i) | \leq \Delta'/8n$ for each node i. We then reset $\Delta' = \max\{| e(i) | : i \in N\}$. As a result, for some node i, $| e(i) | =$

$\Delta' > \Delta'/16n$ and, as in Case 1, within $O(\log n)$ scaling phases, the abundant component containing node i merges into a larger component.

This discussion shows that within $O(\log n)$ scaling phases the algorithm performs one merge operation. Since each merge operation decreases the number of abundant components by one, the algorithm can perform at most n merge operations. Consequently, the number of scaling phases is bounded by $O(n \log n)$. The algorithmic description of the enhanced capacity scaling algorithm implies that the algorithm requires $O(m)$ time per scaling phase plus the time required for the augmentations.

We now obtain a bound on the number of augmentations and the time that they require. The algorithm performs at most n imbalance-property augmentations; it can easily execute each augmentation in $O(m)$ time; thus these augmentations are not a bottleneck step in the algorithm. Next consider the shortest path augmentations. Recall from the proof of Theorem 10.30 that in a scaling phase, we can charge each shortest path augmentation to a root node (which is a large excess or a large-deficit node) or to a nonroot node. Since we can charge each nonroot at most two augmentations over the entire execution of the algorithm, we charge at most $2n$ augmentations to nonroots. Moreover, when we charge an augmentation to a root node i, this node satisfies the condition $|e(i)| \geq (n - 1)\Delta/n$. Theorem 10.33 implies that we will charge at most one augmentation to node i in the following $O(\log n)$ scaling phases before the algorithm performs a merge operation and the component containing node i merges into a larger component. Since the algorithm encounters at most $2n$ different abundant components (n to begin with and n due to merge operations), the total number of shortest path augmentations we can charge to root nodes is at most $O(n \log n)$. Since each shortest path augmentation requires the solution of a shortest path problem with nonnegative arc lengths and requires $S(n, m)$ time, all the shortest path augmentations require a total of $O(n \log n\, S(n, m))$ time. This time dominates the time taken by all other operations performed by the algorithm. Therefore, we have established the assertion of the theorem. ◆

To solve the capacitated minimum cost flow problem, we transform it to the uncapacitated version using the transformation described in Section 2.4. The resulting uncapacitated network has $n' = n + m$ nodes and $m' = 2m$ arcs. The enhanced capacity scaling algorithm will solve the minimum cost flow problem in the transformed network in $O(n' \log n') = O(m \log m) = O(m \log n^2) = O(m \log n)$ scaling phases and will solve a total of $O(n' \log n') = O(m \log n)$ shortest path problems. Each shortest path problem in the uncapacitated network requires $S(n', m')$ time, but using the ideas described in Exercise 4.53 we can improve this time bound to $S(n, m)$. Therefore, the enhanced capacity scaling algorithm can solve the capacitated minimum cost flow problem in $O(m \log n\, S(n, m))$ time. We state this important result as a theorem.

Theorem 10.35. *The enhanced capacity scaling algorithm solves a capacitated minimum cost flow problem in $O(m \log n\, S(n, m))$ time.* ◆

10.8 SUMMARY

In this chapter we continued our study of the minimum cost flow problem by developing several polynomial-time algorithms. The scaling technique is a central theme in almost all the algorithms we have discussed. The algorithms discussed use capacity scaling, cost scaling, or both, or use scaling concepts in their proofs. We discussed six polynomial-time algorithms: (1) the capacity scaling algorithm, (2) the cost scaling algorithm, (3) the double scaling algorithm, (4) the minimum mean cycle-canceling algorithm, (5) the repeated capacity scaling algorithm, and (6) the enhanced capacity scaling algorithm. The first three of these algorithms are weakly polynomial; the other three are strongly polynomial. Figure 10.13 specifies the running times of these algorithms.

The capacity scaling algorithm is possibly the simplest of all the polynomial-time algorithms we have discussed. This algorithm is an improved version of the successive shortest path algorithm discussed in Section 9.7; by augmenting flows along paths with sufficiently large residual capacities, this algorithm is able to decrease the number of augmentations from $O(nU)$ to $O(m \log U)$.

Whereas the capacity scaling algorithm scales the capacities, the cost scaling algorithm scales costs. The algorithm maintains ϵ-optimal flows for decreasing values of ϵ and repeatedly executes an improve-approximation procedure that converts an ϵ-optimal flow into an $\epsilon/2$-optimal flow. The computations performed by the improve-approximation procedure are similar to those performed by the preflow-push algorithm for the maximum flow problem. The double scaling algorithm is the same as the cost scaling algorithm except that it uses a different version of the improve-approximation procedure. The improve-approximation procedure in the cost scaling algorithm performs push/relabel steps; in the double scaling algorithm, this procedure augments flow along paths of sufficiently large residual capacity. Justifying its name, within a cost scaling phase, the double scaling algorithm performs a number of capacity scaling phases.

The minimum mean cycle-canceling algorithm for the minimum cost flow problem is different from all the other algorithms discussed in this chapter. The algorithm is startlingly simple to describe and does not make explicit use of the scaling technique; the proof of the algorithm, however, uses arguments from scaling techniques.

Algorithm	Running time
Capacity scaling algorithm	$O((m \log U)(m + n \log n))$
Cost scaling algorithm	$O(n^3 \log(nC))$
Double scaling algorithm	$O(nm \log U \log(nC))$
Minimum mean cycle-canceling algorithm	$O(n^2m^3 \log n)$
Repeated capacity scaling algorithm	$O((m^2 \log n)(m + n \log n))$
Enhanced capacity scaling algorithm	$O((m \log n)(m + n \log n))$

Figure 10.13 Running times of polynomial-time minimum cost flow algorithms.

This algorithm is a special implementation of the cycle canceling algorithm that we described in Section 9.6; it always augments flow along a minimum mean (negative) cycle in the residual network. To establish that this algorithm is strongly polynomial, we show that (1) when the reduced cost of an arc is *sufficiently large*, the flow on the arc becomes "fixed" (i.e., does not change any more); and (2) within $O(nm \log n)$ iterations, at least one additional arc has a sufficiently large reduced cost so that its value becomes fixed.

If we adopt a similar idea in the capacity scaling algorithm, it also becomes strongly polynomial. We showed that whenever the flow on an arc (i, j) is sufficiently large, we can fix the potentials of nodes i and j with respect to each other. The repeated capacity scaling algorithm applies the capacity scaling algorithm and within $O(\log n)$ scaling phases, it identifies an arc (i, j) with a sufficiently large flow. The algorithm then merges the nodes i and j into a single node and starts from scratch again on the modified minimum cost flow problem. The enhanced capacity scaling algorithm, described next, dramatically improves on the repeated capacity scaling algorithm by observing that whenever we contract an arc, we need not start all over again, but can continue the computations and still contract an additional arc within every $O(\log n)$ scaling phases and use only $O(m \log n)$ augmentations in total. This algorithm does not perform contractions explicitly, but does so implicitly by maintaining zero excesses at the contracted nodes (i.e., nonroot nodes).

REFERENCE NOTES

The following account of polynomial-time minimum cost flow algorithms is fairly brief. The surveys by Ahuja, Magnanti, and Orlin [1989, 1991] and by Goldberg, Tardos, and Tarjan [1989] provide more details concerning the development of this field.

Most of the available (combinatorial) polynomial-time algorithms for the minimum cost flow problems use scaling techniques. Edmonds and Karp [1972] introduced the scaling approach and obtained the first weakly polynomial-time algorithm for the minimum cost flow problem. This algorithm used the capacity scaling technique. The algorithm we presented in Section 10.2, which is a variant of Edmonds and Karp's algorithm, is due to Orlin [1988]. From 1972 to 1984, there was little research on scaling techniques. Since 1985, research employing scaling techniques has been extensive. Researchers now recognize that scaling techniques have great theoretical value as well as potential practical significance. Scaling techniques now yield many of the best (in the worst-case sense) available minimum cost flow algorithms.

Röck [1980] and, independently, Bland and Jensen [1985] suggested a cost scaling technique for the minimum cost flow problem. This approach solves the minimum cost flow problem as a sequence of $O(n \log C)$ maximum flow problems. Goldberg and Tarjan [1987] improved on the running time of Röck's algorithm and solved the minimum cost flow problem by solving "almost" $O(\log(nC))$ maximum flow problems. This approach is based on the concept of ϵ-optimality, which is, independently, due to Bertsekas [1979] and Tardos [1985]. We describe this approach in Section 10.3. Goldberg and Tarjan [1987] have developed several improved implementations of this approach, including the wave implementation presented in

Section 10.3. Their best implementation, which runs in $O(nm \log(n^2/m) \log(nC))$ time, uses Fibonacci heaps and finger search trees. Bertsekas and Eckstein [1988], independently, discovered the wave implementation.

Ahuja, Goldberg, Orlin, and Tarjan [1992] developed the double scaling algorithm described in Section 10.4, which combines capacity and cost scaling. This paper also describes several improved implementations, the best of which runs in $O(nm \log \log U \log(nC))$ time and uses the Fibonacci heap data structure.

When Edmonds and Karp [1972] suggested the first (weakly) polynomial-time algorithm for the minimum cost flow problem, they posed the development of a strongly polynomial-time algorithm as an open challenging problem. Tardos [1985] first settled this problem. Subsequently, Orlin [1984], Fujishige [1986], Galil and Tardos [1986], Goldberg and Tarjan [1987, 1988], Orlin [1988], and Ervolina and McCormick [1990b] developed other strongly polynomial-time algorithms. Currently, the best strongly polynomial-time algorithm is due to Orlin [1988]; it runs in $O((m \log n)(m + n \log n))$ time.

Most of the strongly polynomial-time minimum cost flow algorithm use the ideas of "fixing arc flows" or "fixing node potentials." Tardos [1985] was the first investigator to propose the use of either of these ideas (her algorithm fixes arc flows). The minimum mean cycle-canceling algorithm that we presented in Section 10.5 fixes arc flows; it is due to Goldberg and Tarjan [1988]. Goldberg and Tarjan [1988] also presented several variants of the minimum mean cycle-canceling algorithm with improved worst-case complexity. Orlin [1984] and Fujishige [1986] independently developed the idea of fixing node potentials, which is the "dual" of fixing arc flows. Using this idea, Goldberg, Tardos, and Tarjan [1989] obtained the repeated capacity scaling algorithm that we examined in Section 10.6. The enhanced capacity scaling algorithm, which is due to Orlin [1988], achieves the best strongly polynomial-time for solving the minimum cost flow problem. However, our presentation of the enhanced capacity scaling algorithm in Section 10.7 is based on Plotkin and Tardos' [1990] simplification of Orlin's original algorithm.

Some additional polynomial-time minimum cost flow algorithms include (1) a triple scaling algorithm due to Gabow and Tarjan [1989a], (2) a special implementation of the cycle canceling algorithm developed by Barahona and Tardos [1989], and (3) (its dual approach) a cut canceling algorithm proposed by Ervolina and McCormick [1990a].

Interior point linear programming algorithms are another source of polynomial-time algorithms for the minimum cost flow problem. Among these, the fastest available algorithm, due to Vaidya [1989], solves the minimum cost flow problem in $O(n^{2.5}\sqrt{m}\, K)$ time, with $K = \log n + \log C + \log U$.

Currently, the best available time bound for the minimum cost flow problem is $O(\min\{nm \log(n^2/m) \log(nC), nm (\log \log U) \log(nC), (m \log n)(m + n \log n)\})$; the three bounds in this expression are, respectively, due to Goldberg and Tarjan [1987], Ahuja, Goldberg, Orlin, and Tarjan [1992], and Orlin [1988].

EXERCISES

10.1. Suppose that we want to solve the minimum cost flow problem shown in Figure 10.14(a) by the capacity scaling algorithm. Show the computations for two scaling phases. You may identify the shortest path distances by inspection.

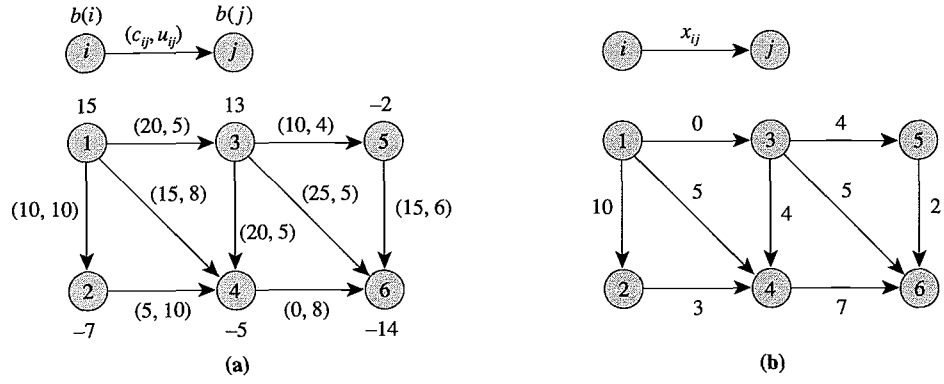


Figure 10.14 Examples for Exercises 10.1 and 10.4.

10.2. In every iteration of the capacity scaling algorithm, we augment flow along a shortest path from a node k with $e(k) \geq \Delta$ to a node l with $e(l) \leq -\Delta$. Suppose that we modify the algorithm as follows: We let node l be any deficit node; that is, we do not necessarily assume that $e(l) \leq -\Delta$. Will this modification affect the worst-case complexity of the capacity scaling algorithm?

10.3. Prove or disprove the following statements.

(a) During the Δ-scaling phase of the capacity scaling algorithm, $| e(i) | \leq 2\Delta$ for each node $i \in N$.

(b) While solving a specific instance of the minimum cost flow problem, the capacity scaling algorithm might perform more augmentations than the successive shortest path algorithm.

10.4. Consider the minimum cost flow problem given in Figure 10.14(a) and the feasible flow x shown in Figure 10.14(b). Starting with $\epsilon = 0$, apply two phases of the cost scaling algorithm.

10.5. Show that if the cost-scaling algorithm finds that arc (i, j) is inadmissible at some stage, this arc remains inadmissible until the algorithm relabels node i.

10.6. Let x and x' be two distinct (feasible) flows in a network. The flow decomposition theorem implies that we can always express x' as x plus the flow along at most m directed cycles $W_1, W_2, \ldots, W_p$ in $G(x)$. For every $1 \leq i \leq p$, let W_i^r denote the directed cycle obtained by reversing each arc in W_i. Show that we can express x as x' plus the flow along the cycles $W_1^r, W_2^r, \ldots, W_p^r$.

10.7. For the cost scaling algorithm, we showed that whenever $\epsilon < 1/n$, any ϵ-optimal flow is 0-optimal. Show that if we multiply all arc costs by $n + 1$, then any flow that is ϵ-optimal flow for the modified problem when $\epsilon \leq 1$ is 0-optimal for the original problem.

10.8. In the cost scaling algorithm, during a relabel operation we increase node potentials by $\epsilon/2$ units. Show that we can increase node potentials by as much as $\epsilon/2 + \min\{c_{ij}^{\pi} : (i, j) \text{ in } G(x) \text{ and } r_{ij} > 0\}$ and still maintain $\epsilon/2$-optimality of the pseudoflow.

10.9. Let x' be a feasible flow of the minimum cost flow problem and let x be a pseudoflow. Show that in the pseudoflow x, for every node v with an excess, there exists a node w with a deficit and a sequence of nodes $v = v_0, v_1, v_2, \ldots, v_l = w$ that satisfies the property that the path $P = v_0 - v_1 - v_2 - \cdots - v_l$ is a directed path in $G(x)$ and its reversal $\overline{P} = v_l - v_{l-1} - \cdots - v_0$ is a directed path in $G(x')$. *(Hint:* This exercise is similar to Exercise 10.6.)

10.10. In this exercise we study the nonscaled version of the cost scaling algorithm.

(a) Modify the algorithm described in Section 10.3 so that it starts with a 0-optimal

pseudoflow, maintains an $1/(n+1)$-optimal pseudoflow at every step, and terminates with an $1/(n+1)$-optimal flow.

(b) Determine the number of relabel operations, the number of saturating and nonsaturating pushes, and the running time of the algorithm. Compare these numbers with those of the cost scaling algorithm.

10.11. In the wave implementation of the cost scaling algorithm described in Section 10.3, we scaled costs by a factor of 2. Suppose, instead, that we scaled costs by a factor of $k \geq 2$. In that case we start with $\Delta = k^{\lfloor \log C \rfloor}$ and decrease ϵ by a factor of k between two consecutive scaling phases. Outline the changes required in the algorithm and determine the number of scaling phases, relabel operations, and saturating and nonsaturating pushes within a scaling phase. For what value of k is the running time minimum?

10.12. Generalized cost scaling algorithm (Goldberg and Tarjan [1987]). As we noted in the text, by using some of the ideas of the minimum mean cycle-canceling algorithm (described in Section 10.5), we can devise a strongly polynomial-time version of the cost scaling algorithm that we described in Section 10.3. The modified algorithm, which we call the *generalized cost scaling algorithm*, is the same as the cost scaling algorithm except that it performs the following additional step after it has called the procedure improve-approximation, but before resetting $\epsilon := \epsilon/2$ (see Figure 10.3).

Additional step: Solve a minimum mean cycle problem to determine the minimum mean cycle cost $\mu(x)$, set $\epsilon = -\mu(x)$, and then determine a set of potential π so that the flow x is ϵ-optimal with respect to π (as described in the proof of Lemma 10.12).

Show that the generalized cost scaling fixes a distinct arc after $O(\log n)$ scaling phases. What is the resulting running time of the algorithm?

10.13. In the double scaling algorithm described in Section 10.4, we scaled costs by a factor of 2. Suppose that as described in Exercise 10.2, we scale costs by a factor of k instead of 2. Show that within a cost scaling phase, the algorithm performs $O(knm)$ retreat steps. How many advance steps does the algorithm perform within a scaling phase? How many scaling phases does it require? For what value of k does the algorithm run in the least time? What is the time bound for this value of k?

10.14. An arc (i, j) in the network $G = (N, A)$ is *critical* if increasing c_{ij} causes the cost of the optimal flow to increase and decreasing c_{ij} causes the cost of the optimal flow to decrease. Does a network always contain a critical arc? Show that we can identify all critical arcs by solving $O(m)$ maximum flow problems. (*Hint*: Use the fact that an arc is critical if it carries a positive flow in every optimal flow.)

10.15. In some minimum cost flow problem, each arc capacity and each supply/demand is a multiple of α and lies in the range $[0, \alpha K]$ for some constant K. Will the algorithms discussed in this chapter run any faster when applied to minimum cost flow problems with this special structure?

10.16. Suppose that in some minimum cost flow problem, each arc cost is a multiple of α and lies in the range $[0, \alpha K]$ for some constant K. Will this special structure permit us to solve the minimum cost flow problem any faster by the cost scaling and double scaling algorithms?

10.17. Minimum cost flows in unit capacity networks. A network is a *unit capacity network* if each arc has a capacity of 1.

(a) What is the running time of the capacity scaling algorithm for unit capacity networks?

(b) What is the running time of the cost scaling algorithm for unit capacity networks? (*Hint*: Will the algorithm make any nonsaturating pushes?)

10.18. Minimum cost flows in bipartite networks. Let $G = (N_1 \cup N_2, A)$ be a bipartite network. Let $n_1 = |N_1| \leq |N_2| = n_2$.

(a) Show that when applied to a bipartite network, the cost scaling algorithm relabels any node $O(n_1)$ times during a scaling phase.

(b) Develop an implementation of the generic cost scaling algorithm that runs in $O(n_1^2 m \log(nC))$ time for bipartite networks. (*Hint*: Generalize the bipartite preflow-push algorithm for the maximum flow problem discussed in Section 8.3.)

10.19. What is the running time of the double scaling algorithm for bipartite networks $G = (N_1 \cup N_2, A)$, assuming that $n_1 = |N_1| \leq |N_2| = n_2$?

10.20. Two minimum cost flow problems P' and P'' are *capacity adjacent* if P'' differs from P' only in one arc capacity and by 1 unit. Given an optimal solution of P', describe an efficient method for solving P''. (*Hint*: Reoptimize by solving a shortest path problem.)

10.21. Two minimum cost flow problems P' and P'' are *cost adjacent* if P'' differs from P' only in one arc cost, and by 1 unit. Given an optimal solution of P', describe an efficient method for solving P''. (*Hint*: Reoptimize by solving a maximum flow problem.)

10.22. Bit scaling of capacities (Röck [1980]). In this capacity scaling algorithm, we consider binary representations of the arc capacities (as described in Section 3.3) and define problem P^k to be the minimum cost flow problem with each arc capacity equal to the k leading bits of the actual capacity. Given an optimal solution of P^k, how would you obtain an optimal solution of P^{k+1} by solving at most m capacity adjacent problems (as defined in Exercise 10.20). Write a pseudocode for the minimum cost flow problem assuming the availability of a subroutine for solving capacity adjacent problems (i.e., solving one from the solution to the other). What is the running time of your algorithm?

10.23. Bit scaling of costs (Röck [1980]). In this cost scaling algorithm, we consider binary representations of the arc costs and define problem P^k to be the minimum cost flow problem with each arc cost equal to the k leading bits of the actual cost. Given an optimal solution of P^k, how would you obtain an optimal solution of P^{k+1} by solving at most m cost adjacent problems (as defined in Exercise 10.21)? Write a pseudocode for the minimum cost flow problem assuming the availability of a subroutine for solving cost adjacent problems (i.e., solving one from the solution to the other). What is the running time of your algorithm?

10.24. Suppose that we define the contraction of an arc as in Section 10.5. Let G^c denote the network of $G = (N, A)$ we obtain when we contract the endpoints of an arc $(k, l) \in A$ into a single node p. In addition, let $G' = (N, A - \{(k, l)\})$. Show that if $\alpha(G)$ denotes the number of (distinct) spanning trees of G, then $\alpha(G) = \alpha(G^c) + \alpha(G')$.

10.25. Constrained maximum flow problem. In the constrained maximum flow problem, we wish to maximize the flow from the source node s to the sink node t subject to an additional linear constraint. Consider the following linear programming formulation of this problem:

$$\text{Maximize} \quad v$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} v & \text{for } i = s \\ 0 & \text{for all } i \in N - \{s,t\} \\ -v & \text{for } i = t, \end{cases}$$

$$0 \leq x_{ij} \leq u_{ij},$$

$$\sum_{(i,j)\in A} c_{ij}x_{ij} \leq D.$$

(a) Let v^* be any integer and let x^* be an optimal solution of a minimum cost flow problem with the objective function $\sum_{(i,j)\in A} c_{ij}x_{ij}$ and with the supply/demand data $b(s) = v^*$, $b(t) = -v^*$, and $b(i) = 0$ for all other nodes. Let $z^* = \sum_{(i,j)\in A} c_{ij}x_{ij}^*$. Show that x^* solves the constrained maximum flow problem when $D = z^*$. Assume that $c_{ij} \geq 0$ for each arc $(i, j) \in A$.

(b) Assume that all of the data in the constrained maximum flow problem are integer. Use the result in part (a) to develop an algorithm for the constrained maximum

flow problem that uses a minimum cost flow algorithm as a subroutine. What is the running time of your algorithm? (*Hint*: Perform binary search on v.)

10.26. In the enhanced capacity scaling algorithm, suppose we maintain an index with each arc that stores whether the arc is an abundant or a nonabundant arc. Suppose further that at some stage the algorithm adds an arc (i, j) to the abundant subgraph. Show how you would perform each of the following operations in $O(m)$ time: (i) identifying the root nodes, i_r and j_r, of the abundant components containing the nodes i and j; (ii) determining whether the nodes i and j belong to the same abundant component; and (iii) identifying a path from node i to j, or vice versa. Using these operations, explain how you would perform a merge operation and the subsequent imbalance-property augmentation in $O(m)$ time. (*Hint*: Observe that each abundant arc can be traversed in either direction because it has sufficient residual capacity in both the directions. Then use the search algorithm described in Section 3.4.)

11

MINIMUM COST FLOWS: NETWORK SIMPLEX ALGORITHMS

. . . seek, and ye shall find.
—The Book of Matthew

Chapter Outline

11.1 INTRODUCTION

The simplex method for solving linear programming problems is perhaps the most powerful algorithm ever devised for solving constrained optimization problems. Indeed, many members of the academic community view the simplex method as not only one of the principal computational engines of applied mathematics, computer science, and operations research, but also as one of the landmark contributions to computational mathematics of this century. The algorithm has achieved this lofty status because of the pervasiveness of its applications throughout many problem domains, because of its extraordinary efficiency, and because it permits us to not only solve problems numerically, but also to gain considerable practical and theoretical insight through the use of sensitivity analysis and duality theory.

Since minimum cost flow problems define a special class of linear programs, we might expect the simplex method to be an attractive solution procedure for solving many of the problems that we consider in this text. Then again, because network flow problems have considerable special structure, we might also ask whether the simplex method could possibly compete with other "combinatorial" methods, such as the many variants of the successive shortest path algorithm, that exploit the underlying network structure. The general simplex method, when implemented in

a way that does not exploit underlying network structure, is not a competitive solution procedure for solving minimum cost flow problems. Fortunately, however, if we interpret the core concepts of the simplex method appropriately as network operations, we can adapt and streamline the method to exploit the network structure of the minimum cost flow problem, producing an algorithm that is very efficient. Our purpose in this chapter is to develop this network-based implementation of the simplex method and show how to apply it to the minimum cost flow problem, the shortest path problem, and the maximum flow problem.

We could adopt several different approaches for presenting this material, and each has its own merits. For example, we could start by describing the simplex method for general linear programming problems and then show how to adapt the method for minimum cost flow problems. This approach has the advantage of placing our development in the broader context of more general linear programs. Alternatively, we could develop the network simplex method directly in the context of network flow problems as a particular type of augmenting cycle algorithm. This approach has the advantage of not requiring any background in linear programming and of building more directly on the concepts that we have developed already. We discuss both points of view. Throughout most of this chapter we adopt the network approach and derive the network simplex algorithm from the first principles, avoiding the use of linear programming in any direct way. Later, in Section 11.11, we show that the network simplex algorithm is an adaptation of the simplex method.

The central concept underlying the network simplex algorithm is the notion of spanning tree solutions, which are solutions that we obtain by fixing the flow of every arc not in a spanning tree either at value zero or at the arc's flow capacity. As we show in this chapter, we can then solve uniquely for the flow on all the arcs in the spanning tree. We also show that the minimum cost flow problem always has at least one optimal spanning tree solution and that it is possible to find an optimal spanning tree solution by "moving" from one such solution to another, at each step introducing one new nontree arc into the spanning tree in place of one tree arc. This method is known as the network simplex algorithm because spanning trees correspond to the so-called basic feasible solutions of linear programming, and the movement from one spanning tree solution to another corresponds to a so-called pivot operation of the general simplex method. In Section 11.11 we make these connections.

In the first three sections of this chapter we examine several fundamental ideas that either motivate the network simplex method or underlie its development. In Section 11.2 we show that the minimum cost flow problem always has at least one spanning tree solution. We also show how the network optimality conditions that we have used repeatedly in previous chapters specialize when applied to any spanning tree solution. In keeping with our practice in previous chapters, we use these conditions to assess whether a candidate solution is optimal and, if not, how to modify it to construct a better spanning tree solution.

To implement the network simplex algorithm efficiently we need to develop a method for representing spanning trees conveniently in a computer so that we can perform the basic operations of the algorithm efficiently and so that we can efficiently manipulate the computer representation of a spanning tree structure from step to step. We describe one such approach in Section 11.3.

In Section 11.4 we show how to compute the arc flows corresponding to any spanning tree and associated node potentials so that we can assess whether the particular spanning tree is optimal. These operations are essential to the network simplex algorithm, and since we need to make these computations repeatedly as we move from one spanning tree to another, we need to be able to implement these operations very efficiently. Section 11.5 brings all these pieces together and describes the network simplex algorithm.

In the context of applying the network simplex algorithm and establishing that the algorithm properly solves any given minimum cost flow problem, we need to address a technical issue known as degeneracy (which occurs when one of the arcs in a spanning tree, like the nontree arcs, has a flow value equal to zero or the arc's flow capacity). In Section 11.6 we describe a very appealing and simple way to modify the basic network simplex algorithm so that it overcomes the difficulties associated with degeneracy.

Since the shortest path and maximum flow problems are special cases of the minimum cost flow problem, the network simplex algorithm applies to these problems as well. In Sections 11.7 and 11.8 we describe these specialized implementations. When applied to the shortest path problem, the network simplex algorithm closely resembles the label-correcting algorithms that we discussed in Chapter 5. When applied to the maximum flow problem, the algorithm is essentially an augmenting path algorithm.

The network simplex algorithm maintains a feasible solution at each step; by moving from one spanning tree solution to another, it eventually finds a spanning tree solution that satisfies the network optimality conditions. Are there other spanning tree algorithms that iteratively move from one infeasible spanning tree solution to another and yet eventually find an optimal solution? In Section 11.9 we describe two such algorithms: a *parametric network simplex algorithm* that satisfies all of the optimality conditions except the mass balance constraints at two nodes, and a *dual network simplex algorithm* that satisfies the mass balance constraints at all the nodes but might violate the arc flow bounds. These algorithms are important because they provide alternative solution strategies for solving minimum cost flow problems; they also illustrate the versatility of spanning tree manipulation algorithms for solving network flow problems.

We next consider a key feature of the optimal spanning tree solutions generated by the network simplex algorithm. In Section 11.10 we show that it is easy to use these solutions to conduct sensitivity analysis: that is, to determine a new solution if we change any cost coefficient or change the capacity of any arc. This type of information is invaluable in practice because problem data are often only approximate and/or because we would like to understand how robust a solution is to changes in the underlying data.

To conclude this chapter we delineate connections between the network simplex algorithm and more general concepts in linear and integer programming. In Section 11.11 we show that the network simplex algorithm is a special case of the simplex method for general linear programs, although streamlined to exploit the special structure of network flow problems. In particular, we show that spanning trees for the network flow problem correspond in a one-to-one fashion with bases of the linear programming formulation of the problem. We also show that each of

the essential steps of the network simplex algorithm, for example, determining node potentials or moving from one spanning tree to another, are specializations of the usual steps of the simplex method for solving linear programs.

As we have noted in Section 9.6, network flow problems satisfy one very remarkable property: They have optimal integral flows whenever the underlying data are integral. In Section 11.12 we show that this integrality result is a special case of a more general result in linear and integer programming. We define a set of linear programming problems with special constraint matrices, known as *unimodular matrices*, and show that these linear programs also satisfy the integrality property. That is, when solved as linear programs with integral data, problems with these specialized constraint matrices always have integer solutions. Since node–arc incidence matrices satisfy the unimodularity property, this integrality property for linear programming is a strict generalization of the integrality property of network flows. This result provides us with another way to view the integrality property of network flows; it is also suggestive of more general results in integer programming and shows how network flow results have stimulated more general investigations in combinatorial optimization and integer programming.

11.2 CYCLE FREE AND SPANNING TREE SOLUTIONS

Much of our development in previous chapters has relied on a simple but powerful algorithmic idea: To generate an improving sequence of solutions to the minimum cost flow problem, we iteratively augment flows along a series of negative cycles and shortest paths. As one of these variants, the network simplex algorithm uses a particular strategy for generating negative cycles. In this section, as a prelude to our discussion of the method, we introduce some basic background material. We begin by examining two important concepts known as *cycle free solutions* and *spanning tree solutions*.

For any feasible solution, x, we say that an arc (i, j) is a *free arc* if $0 < x_{ij} < u_{ij}$ and is a *restricted arc* if $x_{ij} = 0$ or $x_{ij} = u_{ij}$. Note that we can both increase and decrease flow on a free arc while honoring the bounds on arc flows. However, in a restricted arc (i, j) at its lower bound (i.e., $x_{ij} = 0$) we can only increase the flow. Similarly, for flow on a restricted arc (i, j) at its upper bound (i.e., $x_{ij} = u_{ij}$) we can only decrease the flow. We refer to a solution x as a *cycle free solution* if the network contains no cycle composed only of free arcs. Note that in a cycle free solution, we can augment flow on any augmenting cycle in only a single direction since some arc in any cycle will restrict us from either increasing or decreasing that arc's flow. We also refer to a feasible solution x and an associated spanning tree of the network as a *spanning tree solution* if every nontree arc is a restricted arc. Notice that in a spanning tree solution, the tree arcs can be free or restricted. Frequently, when we refer to a spanning tree solution, we do not explicitly identify the associated tree; rather, it will be understood from the context of our discussion.

In this section we establish a fundamental result of network flows: minimum cost flow problems always have optimal cycle free and spanning tree solutions. The network simplex algorithm will exploit this result by restricting its search for an optimal solution to only spanning tree solutions. To illustrate the argument used to prove these results, we use the network example shown in Figure 11.1.

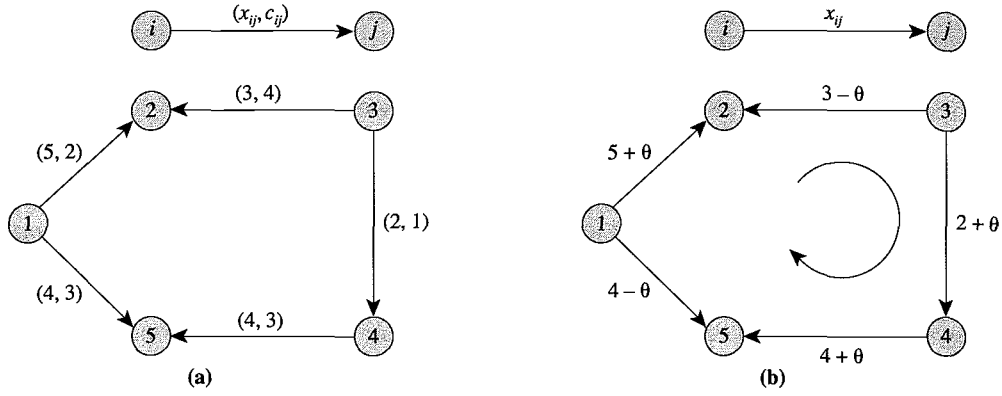


Figure 11.1 Improving flow around a cycle: (a) feasible solution; (b) solution after augmenting θ amount of flow along a cycle.

For the time being let us assume that all arcs are uncapacitated [i.e., $u_{ij} = \infty$ for each $(i, j) \in A$]. The network shown in Figure 11.1 contains positive flow around a cycle. We define the orientation of the cycle as the same as that of arc (4, 5). Let us augment θ units of flow along the cycle in the direction of its orientation. As shown in Figure 11.1, this augmentation increases the flow on arcs along the orientation of the cycle (i.e., forward arcs) by θ units and decreases the flow on arcs opposite to the orientation of the cycle (i.e., backward arcs) by θ units. Also note that the per unit incremental cost for this flow change is the sum of the costs of forward arcs minus the sum of the costs of backward arcs in the cycle, that is,

$$\text{per unit change in cost } \Delta = 2 + 1 + 3 - 4 - 3 = -1.$$

Since augmenting flow in the cycle decreases the cost, we set θ as large as possible while preserving nonnegativity of all arc flows. Therefore, we must satisfy the inequalities $3 - \theta \geq 0$ and $4 - \theta \geq 0$, and hence we set $\theta = 3$. Note that in the new solution (at $\theta = 3$), some arc in the cycle has a flow at value zero, and moreover, the objective function value of this solution is strictly less than the value of the initial solution.

In our example, if we change c_{12} from 2 to 5, the per unit cost of the cycle is $\Delta = 2$. Consequently, to improve the cost by the greatest amount, we would decrease θ as much as possible (i.e., satisfy the restrictions $5 + \theta \geq 0$, $2 + \theta \geq 0$, and $4 + \theta \geq 0$, or $\theta \geq -2$) and again find a lower cost solution with the flow on at least one arc in the cycle at value zero. We can restate this observation in another way: To preserve nonnegativity of all the arc flows, we must select θ in the interval $-2 \leq \theta \leq 3$. Since the objective function depends linearly on θ, we optimize it by selecting $\theta = 3$ or $\theta = -2$, at which point one arc in the cycle has a flow value of zero.

We can extend this observation in several ways:

1. If the per unit cycle cost $\Delta = 0$, we are indifferent to all solutions in the interval $-2 \leq \theta \leq 3$ and therefore can again choose a solution as good as the original one, but with the flow of at least one arc in the cycle at value zero.
2. If we impose upper bounds on the flow (e.g., such as 6 units on all arcs), the

range of flow that preserves feasibility (i.e., the mass balance constraints, lower and upper bounds on flows) is again an interval, in this case $-2 \leq \theta \leq 1$, and we can find a solution as good as the original one by choosing $\theta = -2$ or $\theta = 1$. At these values of θ, the solution is cycle free; that is, some arc on the cycle has a flow either at value zero (at the lower bound) or at its upper bound.

In general, our prior observations apply to any cycle in a network. Therefore, given any initial flow we can apply our previous argument repeatedly, one cycle at a time, and establish the following fundamental result.

Theorem 11.1 (Cycle Free Property). *If the objective function of a minimum cost flow problem is bounded from below over the feasible region, the problem always has an optimal cycle free solution.* ◆

It is easy to convert a cycle free solution into a spanning tree solution. Our results in Section 2.2 show that the free arcs in a cycle free solution define a forest (i.e., a collection of node-disjoint trees). If this forest is a spanning tree, the cycle free solution is already a spanning tree solution. However, if this forest is not a spanning tree, we can add some restricted arcs and produce a spanning tree.

Figure 11.2 illustrates a spanning tree corresponding to a cycle free solution.

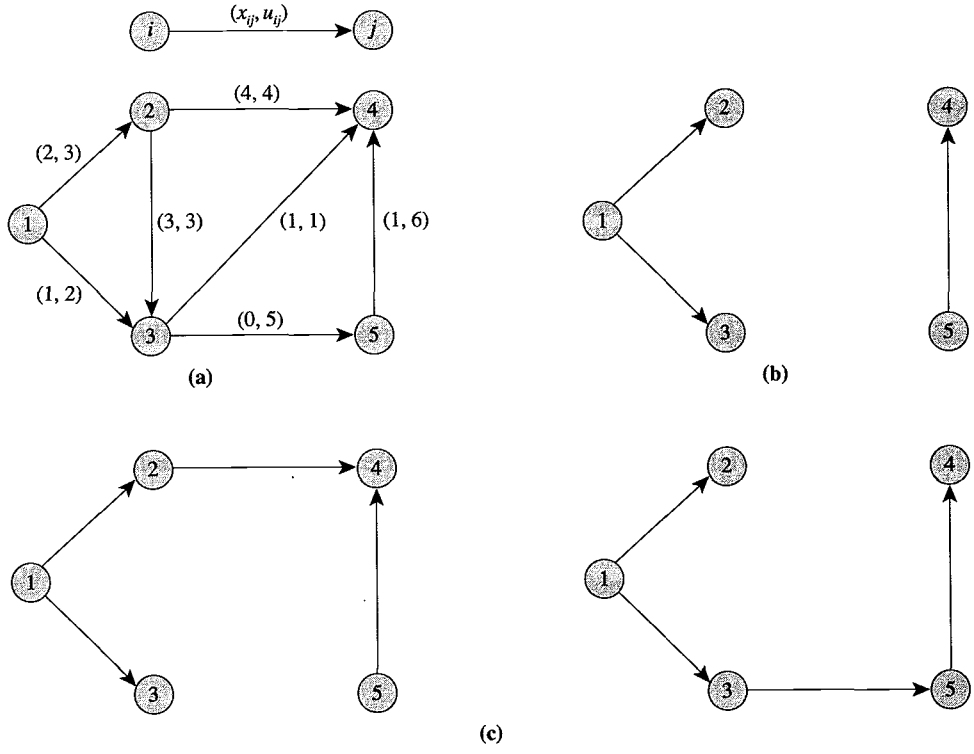


Figure 11.2 Converting a cycle free solution into a spanning tree solution: (a) example network; (b) set of free arcs; (c) 2 spanning tree solutions.

The solution in Figure 11.2(a) is cycle free. Figure 11.2(b) represents the set of free arcs, and Figure 11.2(c) shows two spanning tree solutions corresponding to the cycle free solution. As shown by this example, it might be possible (and often is) to complete the set of free arcs into a spanning tree in several ways. Adding the arc (3, 4) instead of the arc (2, 4) or (3, 5) would produce yet another spanning tree solution. Therefore, a given cycle free solution can correspond to several spanning trees. Nevertheless, since we assume that the underlying network is connected, we can always add some restricted arcs to the free arcs of a cycle free solution to produce a spanning tree, so we have established the following fundamental result:

Theorem 11.2 (Spanning Tree Property). *If the objective function of a minimum cost flow problem is bounded from below over the feasible region, the problem always has an optimal spanning tree solution.* ◆

A spanning tree solution partitions the arc set A into three subsets: (1) **T**, the arcs in the spanning tree; (2) **L**, the nontree arcs whose flow is restricted to value zero; and (3) **U**, the nontree arcs whose flow is restricted in value to the arcs' flow capacities. We refer to the triple (**T**, **L**, **U**) as a *spanning tree structure.*

Just as we can associate a spanning tree structure with a spanning tree solution, we can also obtain a unique spanning tree solution corresponding to a given spanning tree structure (**T**, **L**, **U**). To do so, we set $x_{ij} = 0$ for all arcs $(i, j) \in \mathbf{L}$, $x_{ij} = u_{ij}$ for all arcs $(i, j) \in \mathbf{U}$, and then solve the mass balance equations to determine the flow values for arcs in **T**. In Section 11.4 we show that the flows on the spanning tree arcs are unique. We say that a spanning tree structure is *feasible* if its associated spanning tree solution satisfies all of the arcs' flow bounds. In the special case in which every tree arc in a spanning tree solution is a free arc, we say that the spanning tree is *nondegenerate*; otherwise, we refer to it as a *degenerate* spanning tree. We refer to a spanning tree structure as *optimal* if its associated spanning tree solution is an optimal solution of the minimum cost flow problem. The following theorem states a sufficient condition for a spanning tree structure to be an optimal structure. As shown by our discussion in previous chapters, the reduced costs defined as $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ are useful in characterizing optimal solutions to minimum cost flow problems.

Theorem 11.3 (Minimum Cost Flow Optimality Conditions). *A spanning tree structure* (**T**, **L**, **U**) *is an optimal spanning tree structure of the minimum cost flow problem if it is feasible and for some choice of node potentials* π, *the arc reduced costs* c_{ij}^{π} *satisfy the following conditions:*

$$(a) \quad c_{ij}^{\pi} = 0 \text{ for all } (i, j) \in \mathbf{T}. \tag{11.1a}$$

$$(b) \quad c_{ij}^{\pi} \geq 0 \text{ for all } (i, j) \in \mathbf{L}. \tag{11.1b}$$

$$(c) \quad c_{ij}^{\pi} \leq 0 \text{ for all } (i, j) \in \mathbf{U}. \tag{11.1c}$$

Proof. Let x^* be the solution associated with the spanning tree structure (**T**, **L**, **U**). We know that some set of node potentials π, together with the spanning tree structure (**T**, **L**, **U**), satisfies (11.1).

We need to show that x^* is an optimal solution of the minimum cost flow

problem. In Section 2.4 we showed that minimizing $\sum_{(i,j)\in A} c_{ij}x_{ij}$ is equivalent to minimizing $\sum_{(i,j)\in A} c_{ij}^{\pi}x_{ij}$. The conditions stated in (11.1) imply that for the given node potential π, minimizing $\sum_{(i,j)\in A} c_{ij}^{\pi}x_{ij}$ is equivalent to minimizing the following expression:

$$\text{Minimize} \quad \sum_{(i,j)\in \mathbf{L}} c_{ij}^{\pi}x_{ij} - \sum_{(i,j)\in \mathbf{U}} |\, c_{ij}^{\pi} \,|\, x_{ij}. \tag{11.2}$$

The definition of the solution x^* implies that for any arbitrary solution x, $x_{ij} \geq x_{ij}^*$ for all $(i, j) \in \mathbf{L}$ and $x_{ij} \leq x_{ij}^*$ for all $(i, j) \in \mathbf{U}$. The expression (11.2) implies that the objective function value of the solution x will be greater than or equal to that of x^*. ◆

These optimality conditions have a nice economic interpretation. As we shall see later in Section 11.4, if $\pi(1) = 0$, the equations in (11.1a) imply that $-\pi(k)$ denotes the length of the tree path from node 1 to node k. The reduced cost $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ for a nontree arc $(i, j) \in \mathbf{L}$ denotes the change in the cost of the flow that we realize by sending 1 unit of flow through the tree path from node 1 to node i through the arc (i, j), and then back to node 1 along the tree path from node j to node 1. The condition (11.1b) implies that this circulation of flow is not profitable (i.e., does not decrease cost) for any nontree arc in $\mathbf{L}$. The condition (11.1c) has a similar interpretation.

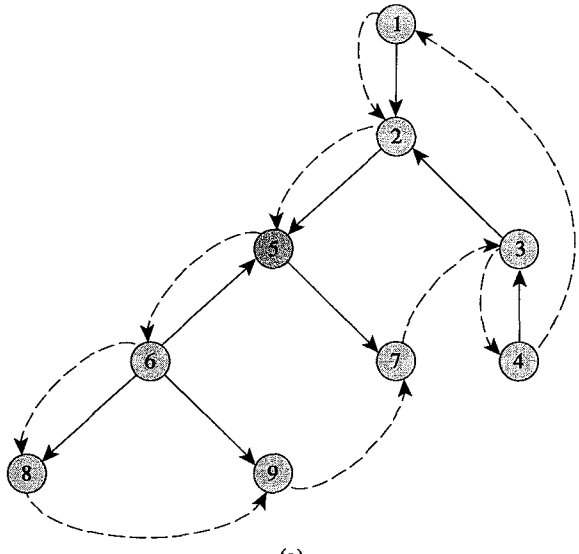
The network simplex algorithm maintains a feasible spanning tree structure and moves from one spanning tree structure to another until it finds an optimal structure. At each iteration, the algorithm adds one arc to the spanning tree in place of one of its current arcs. The entering arc is a nontree arc violating its optimality condition. The algorithm (1) adds this arc to the spanning tree, creating a negative cycle (which might have zero residual capacity), (2) sends the maximum possible flow in this cycle until the flow on at least one arc in the cycle reaches its lower or upper bound, and (3) drops an arc whose flow has reached its lower or upper bound, giving us a new spanning tree structure. Because of its relationship to the primal simplex algorithm for the linear programming problem (see Appendix C), this operation of moving from one spanning tree structure to another is known as a *pivot operation*, and the two spanning trees structures obtained in consecutive iterations are called *adjacent spanning tree structures*. In Section 11.5 we give a detailed description of this algorithm.

11.3 MAINTAINING A SPANNING TREE STRUCTURE

Since the network simplex algorithm generates a sequence of spanning tree solutions, to implement the algorithm effectively, we need to be able to represent spanning trees conveniently in a computer so that the algorithm can perform its basic operations efficiently and can update the representation quickly when it changes the spanning tree. Over the years, researchers have suggested several procedures for maintaining and manipulating a spanning tree structure. In this section we describe one of the more popular representations.

We consider the tree as "hanging" from a specially designated node, called the *root*. Throughout this chapter we assume that node 1 is the root node. Figure

11.3 gives an example of a tree. We associate three indices with each node i in the tree: a predecessor index, $pred(i)$, a depth index $depth(i)$, and a thread index, $thread(i)$.



(a)

i	1	2	3	4	5	6	7	8	9
pred (i)	0	1	2	3	2	5	5	6	6
depth (i)	0	1	2	3	2	3	3	4	4
thread (i)	2	5	4	1	6	8	3	9	7

(b)

Figure 11.3 Example of a tree indices: (a) rooted tree; (b) corresponding tree indices.

Predecessor index. Each node i has a unique path connecting it to the root. The index pred(i) stores the first node in that path (other than node i). For example, the path 9–6–5–2–1 connects node 9 to the root; therefore, pred(9) = 6. By convention, we set the predecessor node of the root node, node 1, equal to zero. Figure 11.3 specifies these indices for the other nodes. Observe that by iteratively using the predecessor indices, we can enumerate the path from any node to the root.

A node j is called a *successor* of node i if pred(j) = i. For example, node 5 has two successors: nodes 6 and 7. A *leaf node* is a node with no successors. In Figure 11.3, nodes 4, 7, 8, and 9 are leaf nodes. The *descendants* of a node i are the node i itself, its successors, successors of its successors, and so on. For example, in Figure 11.3, the elements of node set {5, 6, 7, 8, 9} are the descendants of node 5.

Depth index. We observed earlier that each node i has a unique path connecting it to the root. The index depth(i) stores the number of arcs in that path. For example, since the path 9–6–5–2–1 connects node 9 to the root, depth(9) = 4. Figure 11.3 gives depth indices for all of the nodes in the network.

Thread index. The thread indices define a traversal of a tree, that is, a sequence of nodes that walks or threads its way through the nodes of a tree, starting at the root node, and visiting nodes in a "top-to-bottom" order, and finally returning to the root. We can find thread indices by performing a depth-first search of the tree

as described in Section 3.4 and setting the thread of a node to be the node in the depth-first search encountered just after the node itself. For our example, the depth-first traversal would read 1–2–5–6–8–9–7–3–4–1, so thread(1) = 2, thread(2) = 5, thread(5) = 6, and so on (see the dashed lines in Figure 11.3).

The thread indices provide a particularly convenient means for visiting (or finding) all descendants of a node i. We simply follow the thread starting at that node and record the nodes visited, until the depth of the visited node becomes at least as large as that of node i. For example, starting at node 5, we visit nodes 6, 8, 9, and 7 in order, which are the descendants of node 5 and then visit node 3. Since the depth of node 3 equals that of node 5, we know that we have left the "descendant tree" lying below node 5. We shall see later that finding the descendant tree of a node efficiently is an important step in developing an efficient implementation of the network simplex algorithm.

In the next section we show how the tree indices permit us to compute the feasible solution and the set of node potentials associated with a tree.

11.4 COMPUTING NODE POTENTIALS AND FLOWS

As we noted in Section 11.2, as the network simplex algorithm moves from one spanning tree to the next, it always maintains the condition that the reduced cost of every arc (i, j) in the current spanning tree is zero (i.e. $c_{ij}^{\pi} = 0$). Given the current spanning tree structure (**T**, **L**, **U**), the method first determines values for the node potentials π that will satisfy this condition for the tree arcs. In this section we show how to find these values of the node potentials.

Note that we can set the value of one node potential arbitrarily because adding a constant k to each node potential does not alter the reduced cost of any arc; that is, for any constant k, $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j) = c_{ij} - [\pi(i) + k] + [\pi(j) + k]$. So for convenience, we henceforth assume that $\pi(1) = 0$. We compute the remaining node potentials using the fact that the reduced cost of every spanning tree arc is zero; that is,

$$c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j) = 0 \qquad \text{for every arc } (i, j) \in \mathbf{T}. \tag{11.3}$$

In equation (11.3), if we know one of the node potentials $\pi(i)$ or $\pi(j)$, we can easily compute the other one. Consequently, the basic idea in the procedure is to start at node 1 and fan out along the tree arcs using the thread indices to compute other node potentials. By traversing the nodes using the thread indices, we ensure that whenever the procedure visits a node k, it has already evaluated the potential of its predecessor, so it can compute $\pi(k)$ using (11.3). Figure 11.4 gives a formal statement of the procedure *compute-potentials*.

The numerical example shown in Figure 11.5 illustrates the procedure. We first set $\pi(1) = 0$. The thread of node 1 is 2, so we next examine node 2. Since arc (1, 2) connects node 2 to its predecessor, using (11.3) we find that $\pi(2) = \pi(1) - c_{12} = -5$. We next examine node 5, which is connected to its parent by arc (5, 2). Using (11.3) we obtain $\pi(5) = \pi(2) + c_{52} = -5 + 2 = -3$. In the same fashion we compute the rest of the node potentials; the numbers shown next to each node in Figure 11.5 specify these values.

```
procedure compute-potentials;
begin
    π(1) : = 0;
    j : = thread(1);
    while j ≠ 1 do
    begin
        i : = pred(j);
        if (i, j) ∈ A then π(j) : = π(i) − cij;
        if (j, i) ∈ A then π(j) : = π(i) + cji;
        j : = thread(j);
    end;
end;
```

Figure 11.4 Procedure *compute-potentials*.

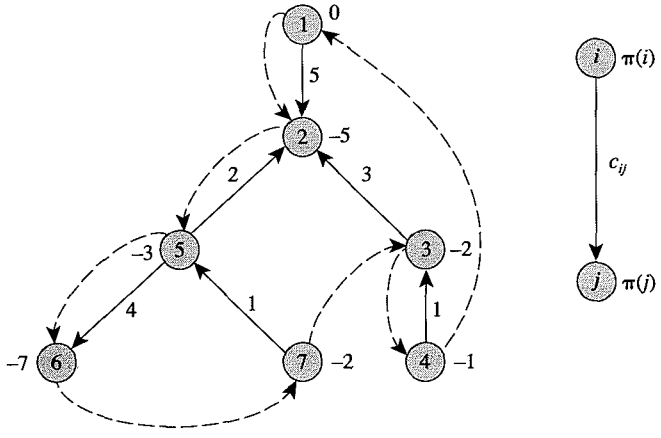


Figure 11.5 Computing node potentials for a spanning tree.

Let P be the tree path in T from the root node 1 to some node k. Moreover, let $\overline{P}$ and $\underline{P}$, respectively, denote the sets of forward and backward arcs in P. Now let us examine arcs in P starting at node 1. The procedure *compute-potentials* implies that $\pi(j) = \pi(i) - c_{ij}$ whenever arc (i, j) is a forward arc in the path, and that $\pi(j) = \pi(i) + c_{ji}$ whenever arc (j, i) is a backward arc in the path. This observation implies that $\pi(k) = \pi(k) - \pi(1) = -\sum_{(i,j)\in\overline{P}} c_{ij} + \sum_{(i,j)\in\underline{P}} c_{ij}$. In other words, $\pi(k)$ is the negative of the cost of sending 1 unit of flow from node 1 to node k along the tree path. Alternatively, $\pi(k)$ is the cost of sending 1 unit of flow from mode k to node 1 along the tree path. The procedure *compute-potentials* requires $O(1)$ time per iteration and performs $(n - 1)$ iterations to evaluate the node potential of each node. Therefore, the procedure runs in $O(n)$ time.

One important consequence of the procedure *compute-potentials* is that the minimum cost flow problem always has integer optimal node potentials whenever all the arc costs are integer. To see this result, recall from Theorem 11.2 that the minimum cost flow problem always has an optimal spanning tree solution. The potentials associated with this tree constitute optimal node potentials, which we can determine using the procedure *compute-potentials*. The description of the procedure *compute-potentials* implies that if all arc costs are integer, node potentials are integer as well (because the procedure performs only additions and subtractions). We refer to this integrality property of optimal node potentials as the *dual integrality property*

since node potentials are the dual linear programming variables associated with the minimum cost flow problem.

Theorem 11.4 (Dual Integrality Property). *If all arc costs are integer, the minimum cost flow problem always has optimal integer node potentials.* ◆

Computing Arc Flows

We next consider the problem of determining the flows on the tree arcs of a given spanning tree structure. To ease our discussion, for the moment let us first consider the *uncapacitated* version of the minimum cost flow problem. We can then assume that all nontree arcs carry zero flow.

If we delete a tree arc, say arc (i, j), from the spanning tree, the tree decomposes into two subtrees. Let T_1 be the subtree containing node i and let T_2 be the subtree containing node j. Note that $\sum_{k \in T_1} b(k)$ denotes the cumulative supply/demand of nodes in T_1 [which must be equal to $-\sum_{k \in T_2} b(k)$ because $\sum_{k \in T_1} b(k) + \sum_{k \in T_2} b(k) = 0$]. In the spanning tree, arc (i, j) is the only arc that connects the subtree T_1 to the subtree T_2, so it must carry $\sum_{k \in T_1} b(k)$ units of flow, for this is the only way to satisfy the mass balance constraints. For example, in Figure 11.6, if we delete arc (1, 2) from the tree, then $T_1 = \{1, 3, 6, 7\}$, $T_2 = \{2, 4, 5\}$, and $\sum_{k \in T_1} b(k) = 10$. Consequently, arc (1, 2) carries 10 units of flow.

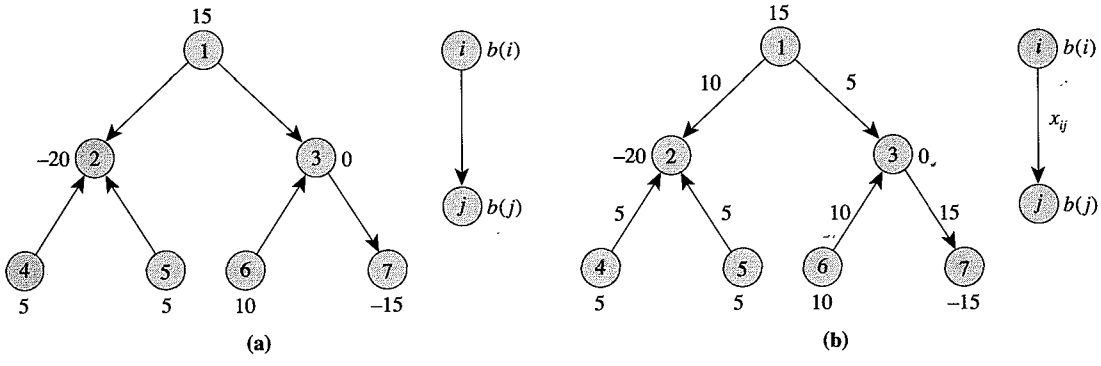


Figure 11.6 Computing flows for a spanning tree.

Using this observation we can devise an efficient method for computing the flows on all the tree arcs. Suppose that (i, j) is a tree arc and that node j is a leaf node [the treatment of the case when (i, j) is a tree arc and node i is a leaf node is similar]. Our observations imply that arc (i, j) must carry $-b(j)$ units of flow. For our example, arc (3, 7) must carry 15 units of flow to satisfy the demand of node 7. Setting the flow on this arc to this value has an effect on the mass balance of its incident nodes: we must subtract 15 units from $b(3)$ and add 15 units to $b(7)$ [which reduces $b(7)$ to zero]. Having determined x_{37}, we can delete arc (3, 7) from the tree and repeat the method on the smaller tree. Notice that we can identify a leaf node in every iteration because every tree has at least two leaf nodes (see Exercise 2.13). Figure 11.7 gives a formal description of this procedure.

```
procedure compute-flows;
begin
    b'(i) : = b(i), for all i ∈ N;
    for each (i, j) ∈ L do set x_ij : = 0;
    T' : = T;
    while T' ≠ {1} do
    begin
        select a leaf node j (other than node 1) in the subtree T';
        i : = pred(j);
        if (i, j) ∈ T' then x_ij : = −b'(j)
        else x_ij : = b'(j);
        add b'(j) to b'(i);
        delete node j and the arc incident to it from T';
    end;
end;
```

Figure 11.7 Procedure *compute-flows*.

This method for computing the flow values assumes that the minimum cost flow problem is uncapacitated. For the capacitated version of the problem, we add the following statement immediately after the first statement [i.e., $b'(i) := b(i)$ for all $i \in N$] in the procedure *compute-flows*. We leave the justification of this modification as an exercise (see Exercise 11.19).

```
for each (i, j) ∈ U do
    set x_ij: = u_ij, subtract u_ij from b'(i) and add u_ij to b'(j);
```

The running time of the procedure *compute-flows* is easy to determine. Clearly, the initialization of flows and modification of supplies/demands $b(i)$ and $b(j)$ for arcs (i, j) in $\mathbf{U}$ requires $O(m)$ time. If we set aside the time to select leaf nodes of $\mathbf{T}$, then each iteration requires $O(1)$ time, resulting in a total of $O(n)$ time. One way of identifying leaf nodes in $\mathbf{T}$ is to select nodes in the reverse order of the thread indices. Note that in the thread traversal, each node appears prior to its descendants (see Property 3.4). We identify the reverse thread traversal of the nodes by examining the nodes in the order dictated by the thread indices, putting all the nodes into a stack in the order of their appearance and then taking them out from the top of the stack one at a time. Therefore, the reverse thread traversal examines each node only after it has examined all of the node's descendants. We have thus established that for the uncapacitated minimum cost flow problem, the procedure *compute-flows* runs in $O(m)$ time. For the capacitated version of the problem, the procedure also requires $O(m)$ time.

We can use the procedure *compute-flows* to obtain an alternative proof of the (primal) integrality property that we stated in Theorem 9.10. Recall from Theorem 11.2 that the minimum cost flow problem always has an optimal spanning tree solution. The flow associated with this tree is an optimal flow and we can determine it using the procedure *compute-flows*. The description of the procedure *compute-flows* implies that if the capacities of all the arcs and the supplies/demands of all the nodes are integer, arc flows are integer as well (because the procedure performs only additions and subtractions). We state this result again because of its importance in network flow theory.

Theorem 11.5 (Primal Integrality Property). *If capacities of all the arcs and supplies/demands of all the nodes are integer, the minimum cost flow problem always has an integer optimal flow.* ◆

In closing this section we observe that every spanning tree structure (**T**, **L**, **U**) defines a unique flow x. If this flow satisfies the flow bounds $0 \leq x_{ij} \leq u_{ij}$ for every arc $(i, j) \in A$, the spanning tree structure is feasible; otherwise, it is infeasible. We refer to the spanning tree **T** as *degenerate* if $x_{ij} = 0$ or $x_{ij} = u_{ij}$ for some arc $(i, j) \in \mathbf{T}$, and *nondegenerate* otherwise. In a nondegenerate spanning tree, $0 < x_{ij} < u_{ij}$ for every tree arc (i, j).

11.5 NETWORK SIMPLEX ALGORITHM

The network simplex algorithm maintains a feasible spanning tree structure at each iteration and successively transforms it into an improved spanning tree structure until it becomes optimal. The algorithmic description in Figure 11.8 specifies the essential steps of the method.

```
algorithm network simplex;
begin
    determine an initial feasible tree structure (T, L, U);
    let x be the flow and π be the node potentials associated with this tree structure;
    while some nontree arc violates the optimality conditions do
    begin
        select an entering arc (k, l) violating its optimality condition;
        add arc (k, l) to the tree and determine the leaving arc (p, q);
        perform a tree update and update the solutions x and π;
    end;
end;
```

Figure 11.8 Network simplex algorithm.

In the following discussion we describe in greater detail how the network simplex algorithm uses tree indices to perform these various steps. This discussion highlights the value of the tree indices in designing an efficient implementation of the algorithm.

Obtaining an Initial Spanning Tree Structure

Our connectedness assumption (i.e., Assumption 9.4 in Section 9.1) provides one way of obtaining an initial spanning tree structure. We have assumed that for every node $j \in N - \{1\}$, the network contains arcs $(1, j)$ and $(j, 1)$, with sufficiently large costs and capacities. We construct the initial tree **T** as follows. We examine each node j, other than node 1, one by one. If $b(j) \geq 0$, we include arc $(1, j)$ in **T** with a flow value of $b(j)$. If $b(j) < 0$, we include arc $(j, 1)$ in **T** with a flow value of $-b(j)$. The set **L** consists of the remaining arcs, and the set **U** is empty. As shown in Section 11.4, we can easily compute the node potentials for this tree using the equations $c_{ij} - \pi(i) + \pi(j) = 0$ for all $(i, j) \in \mathbf{T}$. Recall that we set $\pi(1) = 0$.

If the network does not contain the arcs $(1, j)$ and $(j, 1)$ for each node $j \in$

$N - \{1\}$ (or, we do not wish to add these arcs for some reason), we could construct an initial spanning tree structure by first establishing a feasible flow in the network by solving a maximum flow problem (as described in Application 6.1), and then by converting this solution into a spanning tree solution using the method described in Section 11.2.

Optimality Testing and the Entering Arc

Let (**T**, **L**, **U**) be a feasible spanning tree structure of the minimum cost flow problem, and let π be the corresponding node potentials. To determine whether the spanning tree structure is optimal, we check to see whether the spanning tree structure satisfies the following conditions:

$$c_{ij}^{\pi} \geq 0 \text{ for every arc } (i, j) \in \mathbf{L},$$

$$c_{ij}^{\pi} \leq 0 \text{ for every arc } (i, j) \in \mathbf{U}.$$

If the spanning tree structure satisfies these conditions, it is optimal and the algorithm terminates. Otherwise, the algorithm selects a nontree arc violating the optimality condition to be introduced into the tree. Two types of arcs are *eligible* to enter the tree:

1. Any arc $(i, j) \in \mathbf{L}$ with $c_{ij}^{\pi} < 0$
2. Any arc $(i, j) \in \mathbf{U}$ with $c_{ij}^{\pi} > 0$

For any eligible arc (i, j), we refer to $| c_{ij}^{\pi} |$ as its *violation*. The network simplex algorithm can select *any* eligible arc to enter the tree and still would terminate finitely (with some provisions for dealing with degeneracy, as discussed in Section 11.6). However, different rules for selecting the entering arc produce algorithms with different empirical and theoretical behavior. Many different rules, called *pivot rules*, are possible for choosing the entering arc. The following rules are most widely adopted.

Dantzig's pivot rule. This rule was suggested by George B. Dantzig, the father of linear programming. At each iteration this rule selects an arc with the maximum violation to enter the tree. The motivation for this rule is that the arc with the maximum violation causes the maximum decrease in the objective function per unit change in the value of flow on the selected arc, and hence the introduction of this arc into the spanning tree would cause the maximum decrease per pivot if the average increase in the value of the selected arc were the same for all arcs. Computational results confirm that this choice of the entering arc tends to produce relatively large decreases in the objective function per iteration and, as a result, the algorithm performs fewer iterations than other choices for the pivot rule. However, this rule does have a major drawback: The algorithm must consider every nontree arc to identify the arc with the maximum violation and doing so is very time consuming. Therefore, even though this algorithm generally performs fewer iterations than other implementations, the running time of the algorithm is not attractive.

First eligible arc pivot rule. To implement this rule, we scan the arc list sequentially and select the first eligible arc to enter the tree. In a popular version of this rule, we examine the arc list in a wraparound fashion. For example, in an iteration if we find that the fifth arc in the arc list is the first eligible arc, then in the next iteration we start scanning the arc list from the sixth arc. If we reach the end of the arc list while we are performing some iteration, we continue by examining the arc list from the beginning. One nice feature of this pivot rule is that it quickly identifies the entering arc. The pivot rule does have a counterbalancing drawback: with it, the algorithm generally performs more iterations than it would with other pivot rules because each pivot operation produces a relatively small decrease in the objective function value. The overall effect of this pivot rule on the running time of the algorithm is not very attractive, although the rule does produce a more efficient implementation than Dantzig's pivot rule.

Dantzig's pivot rule and the first pivot rule represent two extreme choices of a pivot rule. The *candidate list pivot rule*, which we discuss next, strikes an effective compromise between these two extremes and has proven to be one of the most successful pivot rules in practice. This rule also offers sufficient flexibility for fine tuning to special circumstances.

Candidate list pivot rule. When implemented with this rule, the algorithm selects the entering arc using a two-phase procedure consisting of *major iterations* and *minor iterations*. In a major iteration we construct a *candidate list* of eligible arcs. Having constructed this list, we then perform a number of minor iterations; in each of these iterations, we select an eligible arc from the candidate list with the maximum violation.

In a major iteration we construct the candidate list as follows. We first examine arcs emanating from node 1 and add eligible arcs to the candidate list. We repeat this process for nodes 2, 3, . . . , until either the list has reached its maximum allowable size or we have examined all the nodes. The next major iteration begins with the node where the previous major iteration ended and examines nodes in a wraparound fashion.

Once the algorithm has formed the candidate list in a major iteration, it performs a number of minor iterations. In a minor iteration, the algorithm scans all the arcs in the candidate list and selects an arc with the maximum violation to enter the tree. As we scan the arcs, we update the candidate list by removing those arcs that are no longer eligible (due to changes in the node potentials). Once the candidate list becomes empty or we have reached a specified limit on the number of minor iterations to be performed within each major iteration, we rebuild the candidate list by performing another major iteration.

Notice that the candidate list approach offers considerable flexibility for fine tuning to special problem classes. By setting the maximum allowable size of the candidate list appropriately and by specifying the number of minor iterations to be performed within a major iteration, we can obtain numerous different pivot rules. In fact, Dantzig's pivot rule and the first eligible pivot rule are special cases of the candidate list pivot rule (see Exercise 11.20).

In the preceding discussion, we described several important pivot rules. In the reference notes, we supply references for other pivot rules. Our next topic of study

is deciding how to choose the arc that leaves the spanning tree structure at each step of the network simplex algorithm.

Leaving Arc

Suppose that we select arc (k, l) as the entering arc. The addition of this arc to the tree **T** creates exactly one cycle W, which we refer to as the *pivot cycle*. The pivot cycle consists of the unique path in the tree **T** from node k to node l, together with arc (k, l). We define the orientation of the cycle W as the same as that of (k, l) if $(k, l) \in \mathbf{L}$ and opposite the orientation of (k, l) if $(k, l) \in \mathbf{U}$. Let $\overline{W}$ and $\underline{W}$ denote the sets of *forward arcs* (i.e., those along the orientation of W) and *backward arcs* (those opposite to the orientation of W) in the pivot cycle. Sending additional flow around the pivot cycle W in the direction of its orientation strictly decreases the cost of the current solution at the per unit rate of $| c_{kl}^{\pi} |$. We augment the flow as much as possible until one of the arcs in the pivot cycle reaches its lower or upper bound. Notice that augmenting flow along W increases the flow on forward arcs and decreases the flow on backward arcs. Consequently, the maximum flow change δ_{ij} on an arc $(i, j) \in W$ that satisfies the flow bound constraints is

$$\delta_{ij} = \begin{cases} u_{ij} - x_{ij} & \text{if } (i, j) \in \overline{W} \\ x_{ij} & \text{if } (i, j) \in \underline{W} \end{cases}$$

To maintain feasibility, we can augment $\delta = \min\{\delta_{ij} : (i, j) \in W\}$ units of flow along W. We refer to any arc $(i, j) \in W$ that defines δ (i.e., for which $\delta = \delta_{ij}$) as a *blocking arc*. We then augment δ units of flow and select an arc (p, q) with $\delta_{pq} = \delta$ as the leaving arc, breaking ties arbitrarily. We say that a pivot iteration is a *nondegenerate iteration* if $\delta > 0$ and is a *degenerate iteration* if $\delta = 0$. A degenerate iteration occurs only if **T** is a degenerate spanning tree. Observe that if two arcs tie while determining the value of δ, the next spanning tree will be degenerate.

The crucial step in identifying the leaving arc is to identify the pivot cycle. If $P(i)$ denotes the unique path in the tree from any node i to the root node, this cycle consists of the arcs $\{(k, l)\} \cup P(k) \cup P(l) - (P(k) \cap P(l))$. In other words, W consists of the arc (k, l) and the disjoint portions of $P(k)$ and $P(l)$. Using the predecessor indices alone permits us to identify the cycle W as follows. First, we designate all the nodes in the network as unmarked. We then start at node k and, using the predecessor indices, trace the path from this node to the root and mark all the nodes in this path. Next we start at node l and trace the predecessor indices until we encounter a marked node, say w. The node w is the first common ancestor of nodes k and l; we refer to it as the *apex* of cycle W. The cycle W contains the portions of the paths $P(k)$ and $P(l)$ up to node w, together with the arc (k, l). This method identifies the cycle W in $O(n)$ time and so is efficient. However, it has the drawback of backtracking along those arcs of $P(k)$ that are not in W. If the pivot cycle lies "deep in the tree," far from its root, then tracing the nodes back to the root will be inefficient. Ideally, we would like to identify the cycle W in time proportional to $| W |$. The simultaneous use of depth and predecessor indices, as indicated in Figure 11.9, permits us to achieve this goal.

This method scans the arcs in the pivot cycle W twice. During the first scan, we identify the apex of the cycle and also identify the maximum possible flow that

```
procedure identify-cycle;
begin
    i : = k and j : = l;
    while i ≠ j do
    begin
        if depth(i) > depth( j) then i : = pred(i)
        else if depth ( j) > depth (i) then j : = pred( j)
            else i : = pred(i) and j : = pred( j);
    end;
end;
```

Figure 11.9 Procedure for identifying the pivot cycle.

can be augmented along W. In the second scan, we augment the flow. The entire flow change operation requires $O(n)$ time in the worst case, but typically it examines only a small subset of nodes (and arcs).

Updating the Tree

When the network simplex algorithm has determined a leaving arc (p, q) for a given entering arc (k, l), it updates the tree structure. If the leaving arc is the same as the entering arc, which would happen when $\delta = \delta_{kl} = u_{kl}$, the tree does not change. In this instance the arc (k, l) merely moves from the set **L** to the set **U**, or vice versa. If the leaving arc differs from the entering arc, the algorithm must perform more extensive changes. In this instance the arc (p, q) becomes a nontree arc at its lower or upper bound, depending on whether (in the updated flow) $x_{pq} = 0$ or $x_{pq} = u_{pq}$. Adding arc (k, l) to the current spanning tree and deleting arc (p, q) creates a new spanning tree.

For the new spanning tree, the node potentials also change; we can update them as follows. The deletion of the arc (p, q) from the previous tree partitions the set of nodes into two subtrees, one, T_1, containing the root node, and the other, T_2, not containing the root node. Note that the subtree T_2 hangs from node p or node q. The arc (k, l) has one endpoint in T_1 and the other in T_2. As is easy to verify, the conditions $\pi(1) = 0$ and $c_{ij} - \pi(i) + \pi(j) = 0$ for all arcs in the new tree imply that the potentials of nodes in the subtree T_1 remain unchanged, and the potentials of nodes in the subtree T_2 change by a constant amount. If $k \in T_1$ and $l \in T_2$, all the node potentials in T_2 increase by $-c^{\pi}_{kl}$; if $l \in T_1$ and $k \in T_2$, they increase by the amount c^{π}_{kl}. Using the thread and depth indices, the method described in Figure 11.10 updates the node potentials quickly.

```
procedure update-potentials;
begin
    if q ∈ T2 then y : = q else y : = p;
    if k ∈ T1 then change : = −c^π_kl else change : = c^π_kl;
    π(y) : = π(y) + change;
    z : = thread(y);
    while depth(z) > depth(y) do
    begin
        π(z) : = π(z) + change;
        z : = thread(z);
    end;
end;
```

Figure 11.10 Updating node potentials in a pivot operation.

The final step in the updating of the tree is to recompute the various tree indices. This step is rather involved and we refer the reader to the references given in reference notes for the details. We do point out, however, that it is possible to update the tree indices in $O(n)$ time. In fact, the time required to update the tree indices is $O(|W| + \min\{|T_1|, |T_2|\})$, which is typically much less than n.

Termination

The network simplex algorithm, as just described, moves from one feasible spanning tree structure to another until it obtains a spanning tree structure that satisfies the optimality condition (11.1). If each pivot operation in the algorithm is nondegenerate, it is easy to show that the algorithm terminates finitely. Recall that $|c_{kl}^{\pi}|$ is the net decrease in the cost per unit flow sent around the pivot cycle W. After a nondegenerate pivot (for which $\delta > 0$), the cost of the new spanning tree structure is $\delta|c_{kl}^{\pi}|$ units less than the cost of the previous spanning tree structure. Since any network has a finite number of spanning tree structures and every spanning tree structure has a unique associated cost, the network simplex algorithm will encounter any spanning tree structure at most once and hence will terminate finitely. Degenerate pivots, however, pose a theoretical difficulty: The algorithm might not terminate finitely unless we perform pivots carefully. In the next section we discuss a special implementation, called the *strongly feasible spanning tree implementation*, that guarantees finite convergence of the network simplex algorithm even for problems that are degenerate.

We use the example in Figure 11.11(a) to illustrate the network simplex algorithm. Figure 11.11(b) shows a feasible spanning tree solution for the problem. For this solution, $\mathbf{T} = \{(1, 2), (1, 3), (2, 4), (2, 5), (5, 6)\}$, $\mathbf{L} = \{(2, 3), (5, 4)\}$, and $\mathbf{U} = \{(3, 5), (4, 6)\}$. In this solution, arc (3, 5) has a positive violation, which is 1 unit. We introduce this arc into the tree creating a cycle whose apex is node 1. Since arc (3, 5) is at its upper bound, the orientation of the cycle is opposite to that of arc (3, 5). The arcs (1, 2) and (2, 5) are forward arcs in the cycle and arcs (3, 5) and (1, 3) are backward arcs. The maximum increase in flow permitted by the arcs (3, 5), (1, 3), (1, 2), and (2, 5) is, respectively, 3, 3, 2, and 1 units. Consequently, $\delta = 1$ and we augment 1 unit of flow along the cycle. The augmentation increases the flow on arcs (1, 2) and (2, 5) by one unit and decreases the flow on arcs (1, 3) and (3, 5) by one unit. Arc (2, 5) is the unique blocking arc and so we select it to leave the tree. Dropping arc (2, 5) from the tree produces two subtrees: T_1 consisting of nodes 1, 2, 3, 4 and T_2 consisting of nodes 5 and 6. Introducing arc (3, 5), we again obtain a spanning tree, as shown in Figure 11.11(c). Notice that in this spanning tree, the node potentials of nodes 5 and 6 are 1 unit less than that in the previous spanning tree.

In the feasible spanning tree solution shown in Figure 11.11(c), $\mathbf{L} = \{(2, 3), (5, 4)\}$ and $\mathbf{U} = \{(2, 5), (4, 6)\}$. In this solution, arc (4, 6) is the only eligible arc: its violation equals 1 unit. Therefore, we introduce arc (4, 6) into the tree. Figure 11.11(c) shows the resulting cycle and its orientation. We can augment 1 unit of additional flow along the orientation of this cycle. Sending this flow, we find that arc (3, 5) is a blocking arc, so we drop this arc from the current spanning tree. Figure 11.11(d)

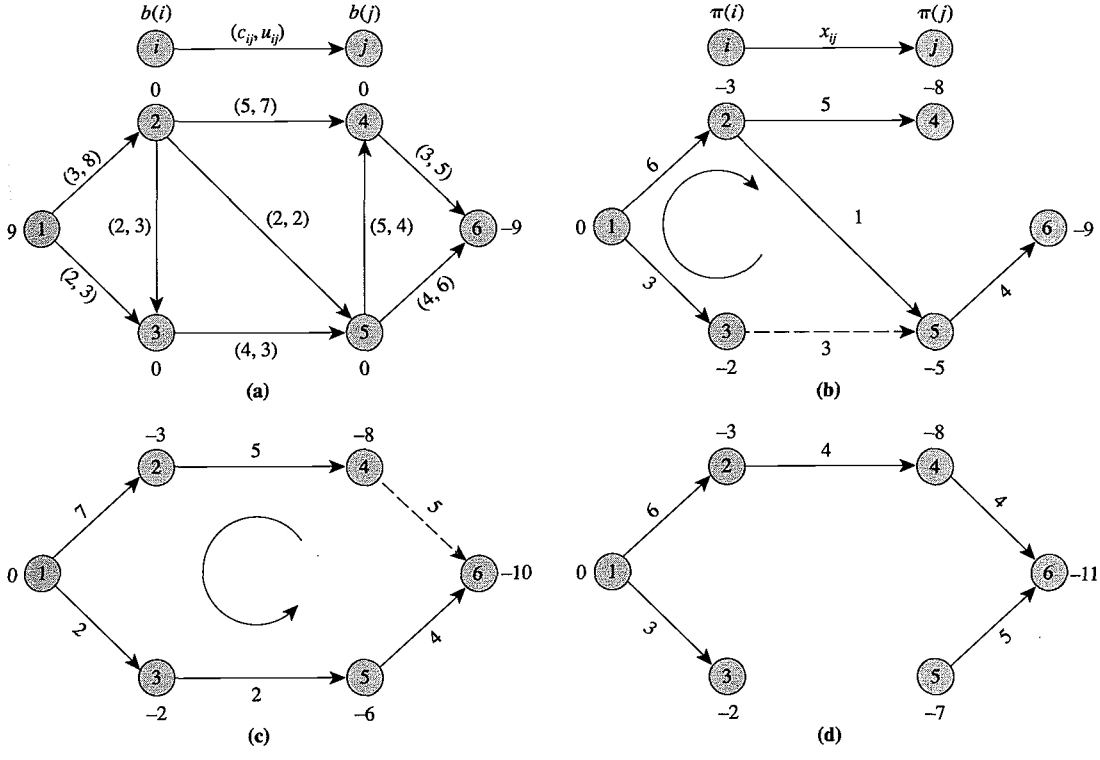


Figure 11.11 Numerical example for the network simplex algorithm.

shows the new spanning tree. As the reader can verify, this solution has no eligible arc, and thus the network simplex algorithm terminates with this solution.

11.6 STRONGLY FEASIBLE SPANNING TREES

The network simplex algorithm does not necessarily terminate in a finite number of iterations unless we impose some additional restriction on the choice of the entering and leaving arcs. Very small network examples show that a poor choice leads to *cycling* (i.e., an infinite repetitive sequence of degenerate pivots). Degeneracy in network problems is not only a theoretical issue, but also a practical one. Computational studies have shown that as many as 90% of the pivot operations in commonplace networks can be degenerate. As we show next, by maintaining a special type of spanning tree, called a *strongly feasible spanning tree*, the network simplex algorithm terminates finitely; moreover, it runs faster in practice as well.

Let (**T**, **L**, **U**) be a spanning tree structure for a minimum cost flow problem with integral data. As before, we conceive of a spanning tree as a tree hanging from the root node. The tree arcs are either *upward pointing* (toward the root) or are *downward pointing* (away from the root). We now state two alternate definitions of a strongly feasible spanning tree.

1. *Strongly feasible spanning tree.* A spanning tree **T** is *strongly feasible* if every tree arc with zero flow is upward pointing and every tree arc whose flow equals its capacity is downward pointing.
2. *Strongly feasible spanning tree.* A spanning tree **T** is *strongly feasible* if we can send a positive amount of flow from any node to the root along the tree path without violating any flow bound.

If a spanning tree **T** is strongly feasible, we also say that the spanning tree structure (**T**, **L**, **U**) is strongly feasible.

It is easy to show that the two definitions of the strongly feasible spanning trees are equivalent (see Exercise 11.24). Figure 11.12(a) gives an example of a strongly feasible spanning tree, and Figure 11.12(b) illustrates a feasible spanning tree that is not strongly feasible. The spanning tree shown in Figure 11.12(b) fails to be strongly feasible because arc (3, 5) carries zero flow and is downward pointing. Observe that in this spanning tree, we cannot send any additional flow from nodes 5 and 7 to the root along the tree path.

To implement the network simplex algorithm so that it always maintains a strongly feasible spanning tree, we must first find an initial strongly feasible spanning tree. The method described in Section 11.5 for constructing the initial spanning tree structure always gives such a spanning tree. Note that a nondegenerate spanning tree is always strongly feasible; a degenerate spanning tree might or might not be strongly feasible. The network simplex algorithm creates a degenerate spanning tree from a nondegenerate spanning tree whenever two or more arcs are qualified as

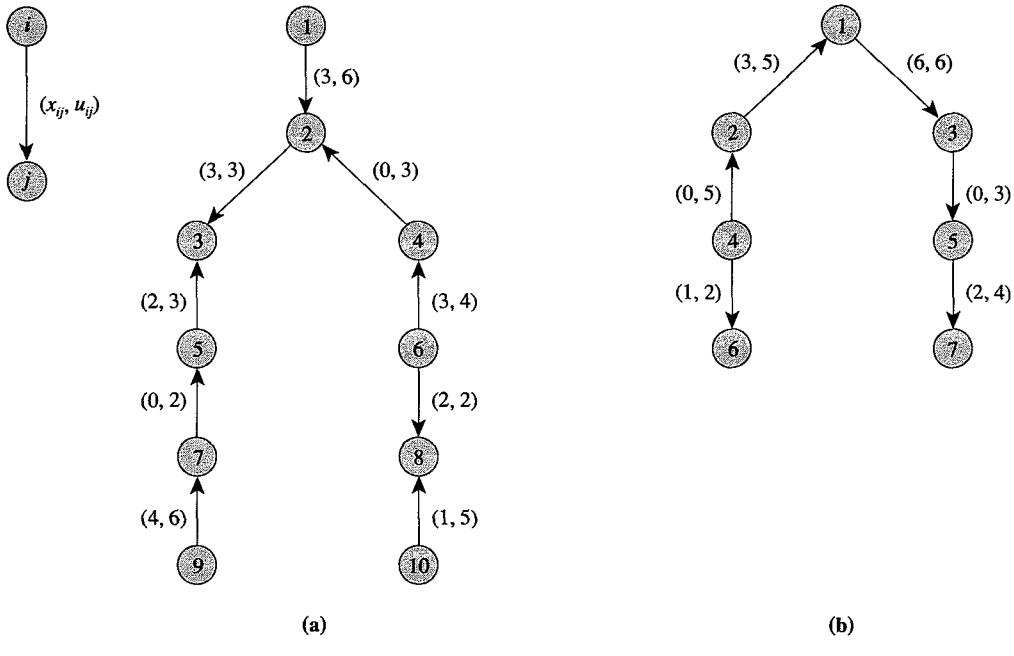


Figure 11.12 Feasible spanning trees: (a) strongly feasible; (b) nonstrongly feasible.

leaving arcs and we drop only one of these. Therefore, the algorithm needs to select the leaving arc carefully so that the next spanning tree is strongly feasible.

Suppose that we have a strongly feasible spanning tree and, during a pivot operation, arc (k, l) enters the spanning tree. We first consider the case when (k, l) is a nontree arc at its lower bound. Suppose that W is the pivot cycle formed by adding arc (k, l) to the spanning tree and that node w is the apex of the cycle W; that is, w is the first common ancestor of nodes k and l. We define the orientation of the cycle W as compatible with that of arc (k, l). After augmenting δ units of flow along the pivot cycle, the algorithm identifies the *blocking arcs* [i.e., those arcs (i, j) in the cycle that satisfy $\delta_{ij} = \delta$]. If the blocking arc is unique, we select it to leave the spanning tree. If the cycle contains more than one blocking arc, the next spanning tree will be degenerate (i.e., some tree arcs will be at their lower or upper bounds). In this case the algorithm selects the leaving arc in accordance with the following rule.

Leaving Arc Rule. *Select the leaving arc as the last blocking arc encountered in traversing the pivot cycle W along its orientation starting at the apex w.*

To illustrate the leaving arc rule, we consider a numerical example. Figure 11.13 shows a strongly feasible spanning tree for this example. Let (9, 10) be the entering arc. The pivot cycle is 10–8–6–4–2–3–5–7–9–10 and the apex is node 2. This pivot is degenerate because arcs (2, 3) and (7, 5) block any additional flow in the pivot cycle. Traversing the pivot cycle starting at node 2, we encounter arc (7, 5) later than arc (2, 3); so we select arc (7, 5) as the leaving arc.

We show that the leaving arc rule guarantees that in the next spanning tree every node in the cycle W can send a positive amount of flow to the root node. Let

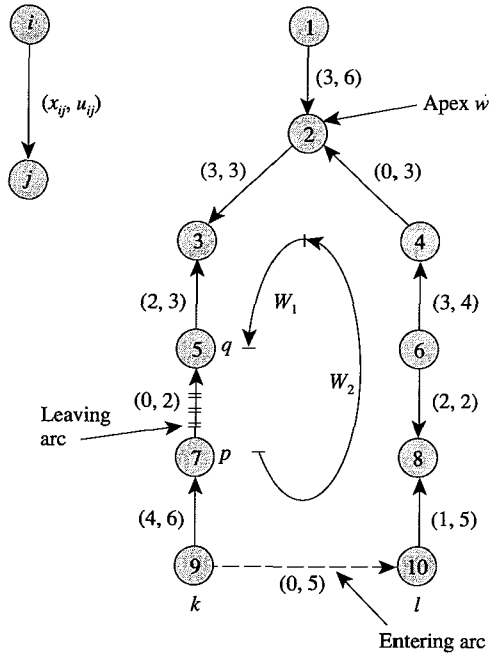


Figure 11.13 Selecting the leaving arc.

(p, q) be the arc selected by the leaving arc rule. Let W_1 be the segment of the cycle W between the apex w and arc (p, q) when we traverse the cycle along its orientation. Let $W_2 = W - W_1 - \{(p, q)\}$. Define the orientation of segments W_1 and W_2 as compatible with the orientation of W. See Figure 11.13 for an illustration of the segments W_1 and W_2. We use the following property about the nodes in the segment W_2.

Property 11.6. *Each node in the segment W_2 can send a positive amount of flow to the root in the next spanning tree.*

This observation follows from the fact that arc (p, q) is the last blocking arc in W; consequently, no arc in W_2 is blocking and every node in this segment can send a positive amount of flow to the root via node w along the orientation of W_2. Note that if the leaving arc does not satisfy the leaving arc rule, no node in the segment W_2 can send a positive amount of flow to the root; therefore, the next spanning tree will not be strongly feasible.

We next focus on the nodes contained in the segment W_1.

Property 11.7. *Each node in the segment W_1 can send a positive amount of flow to the root in the next spanning tree.*

We prove this observation by considering two cases. If the previous pivot was a nondegenerate pivot, the pivot augmented a positive amount of flow δ along the arcs in W_1; consequently, after the augmentation, every node in the segment W_1 can send a positive amount of flow back to the root opposite to the orientation of W_1 via the apex node w (each node can send at least δ units to the apex and then at least some of this flow to the root since the previous spanning tree was strongly feasible). If the previous pivot was a degenerate pivot, W_1 must be contained in the segment of W between node w and node k because the property of strong feasibility implies that every node on the path from node l to node w can send a positive amount of flow to the root before the pivot, and thus no arc on this path can be a blocking arc in a degenerate pivot. Now observe that before the pivot, every node in W_1 could send a positive amount of flow to the root, and therefore since the pivot does not change flow values, every node in W_1 must be able to send a positive amount of flow to the root after the pivot as well. This conclusion completes the proof that in the next spanning tree every node in the cycle W can send a positive amount of flow to the root node.

We next show that in the next spanning tree, nodes not belonging to the cycle W can also send a positive amount of flow to the root. In the previous spanning tree (before the augmentation), every node j could send a positive amount of flow to the root and if the tree path from node j does not pass through the cycle W, the same path is available to carry a positive amount of flow in the next spanning tree. If the tree path from node j does pass through the cycle W, the segment of this tree path to the cycle W is available to carry a positive amount of flow in the next spanning tree and once a positive amount of flow reaches the cycle W, then, as shown earlier, we can send it (or some of it) to the root node. This conclusion completes the proof that the next spanning tree is strongly feasible.

We now establish the finiteness of the network simplex algorithm. Since we have previously shown that each nondegenerate pivot strictly decreases the objective function value, the number of nondegenerate pivots is finite. The algorithm can, however, also perform degenerate pivots. We will show that the number of successive degenerate pivots between any two nondegenerate pivots is finitely bounded. Suppose that arc (k, l) enters the spanning tree at its lower bound and in doing so it defines a degenerate pivot. In this case, the leaving arc belongs to the tree path from node k to the apex w. Now observe from Section 11.5 that node k lies in the subtree T_2 and the potentials of all nodes in T_2 change by an amount c_{kl}^{π}. Since $c_{kl}^{\pi} < 0$, this degenerate pivot strictly decreases the sum of all node potentials (which by our prior assumption is integral). Since no node potential can fall below $-nC$, the number of successive degenerate pivots is finite.

So far we have assumed that the entering arcs are always at their lower bounds. If the entering arc (k, l) is at its upper bound, we define the orientation of the cycle W as opposite to the orientation of arc (k, l). The criteria for selecting the leaving arc remains unchanged—the leaving arc is the last blocking arc encountered in traversing W along its orientation starting at the apex w. In this case node l is contained in the subtree T_2, and thus after the pivot, the potentials of all the nodes T_2 decrease by the amount $c_{kl}^{\pi} > 0$; consequently, the pivot again decreases the sum of the node potentials.

11.7 NETWORK SIMPLEX ALGORITHM FOR THE SHORTEST PATH PROBLEM

In this section we see how the network simplex algorithm specializes when applied to the shortest path problem. The resulting algorithm bears a close resemblance to the label-correcting algorithms discussed in Chapter 5. In this section we study the version of the shortest path problem in which we wish to determine shortest paths from a given source node s to all other nodes in a network. In other words, the problem is to send 1 unit of flow from the source to every other node along minimum cost paths. We can formulate this version of the shortest path problem as the following minimum cost flow model:

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

subject to

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

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

If the network contains a negative (cost)-directed cycle, this linear programming formulation would have an unbounded solution since we could send an infinite amount of flow along this cycle without violating any of the constraints (because the arc flows have no upper bounds). The network simplex algorithm we describe is capable of detecting the presence of a negative cycle, and if the network contains no such cycle, it determines the shortest path distances.

Like other minimum cost flow problems, the shortest path problem has a spanning tree solution. Because node s is the only source node and all the other nodes are demand nodes, the tree path from the source node to every other node is a directed path. This observation implies that the spanning tree must be a *directed out-tree rooted at node s* (see Figure 11.14 and the discussion in Section 4.3). As before, we store this tree using predecessor, depth, and thread indices. In a directed out-tree, every node other than the source has exactly one incoming arc but could have several outgoing arcs. Since each node except node s has unit demand, the flow of arc (i, j) is $| D(j) |$. [Recall that $D(j)$ is the set of descendants of node j in the spanning tree and, by definition, this set includes node j.] Therefore, every tree of the shortest path problem is nondegenerate, and consequently, the network simplex algorithm will never perform degenerate pivots.

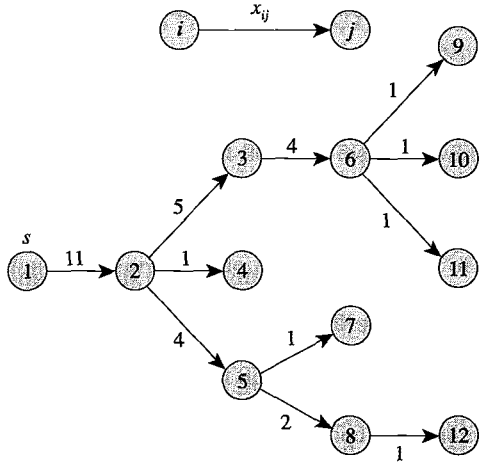


Figure 11.14 Directed out-tree rooted at the source.

Any spanning tree for the shortest path problem contains a unique directed path from node s to every other node. Let $P(k)$ denote the path from node s to node k. We obtain the node potentials corresponding to the tree $\mathbf{T}$ by setting $\pi(s) = 0$ and then using the equation $c_{ij} - \pi(i) + \pi(j) = 0$ for each arc $(i, j) \in \mathbf{T}$ by fanning out from node s (see Figure 11.15). The directed out-tree property of the spanning tree implies that $\pi(k) = -\sum_{(i,j)\in P(k)} c_{ij}$. Thus $\pi(k)$ is the negative of the length of the path $P(k)$.

Since the variables in the minimum cost flow formulation of the shortest path problem have no upper bounds, every nontree arc is at its lower bound. The algorithm selects a nontree arc (k, l) with a negative reduced cost to introduce into the spanning tree. The addition of arc (k, l) to the tree creates a cycle which we orient in the same direction as arc (k, l). Let w be the apex of this cycle. (See Figure 11.16 for an illustration.) In this cycle, every arc from node l to node w is a backward arc and every arc from node w to node k is a forward arc. Consequently, the leaving arc would lie in the segment from node l to node w. In fact, the leaving arc would be the arc (pred(l), l) because this arc has the smallest flow value among all arcs in the segment from node l to node w. The algorithm would then increase the potentials of nodes in the subtree rooted at the node l by an amount $| c_{kl}^{\pi} |$, update the tree

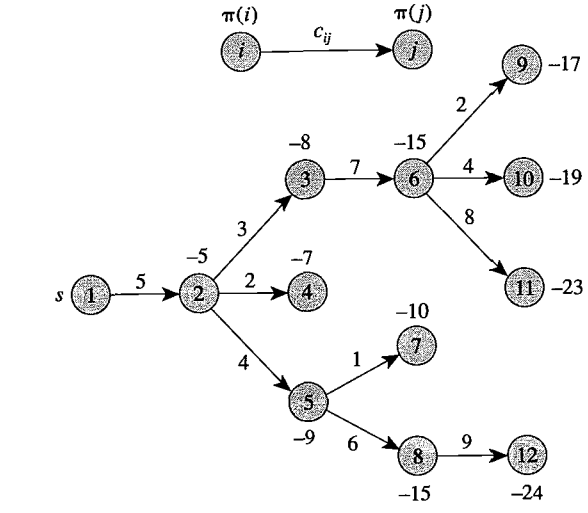


Figure 11.15 Computing node potentials.

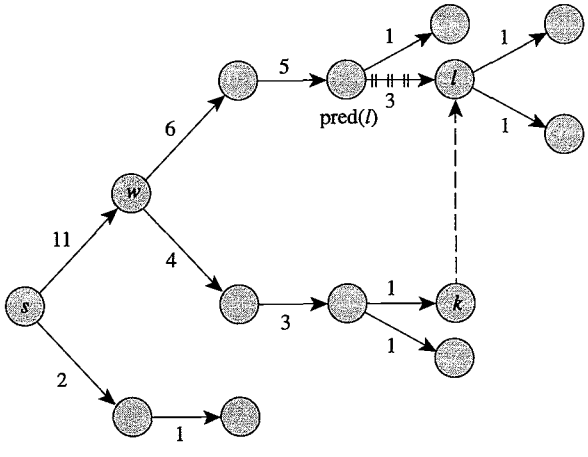


Figure 11.16 Selecting the leaving arc.

indices, and repeat the computations until all nontree arcs have nonnegative reduced costs. When the algorithm terminates, the final tree would be a shortest path tree (i.e., a tree in which the directed path from node s to every other node is a shortest path).

Recall that in implementing the network simplex algorithm for the minimum cost flow problem, we maintained flow values for all the arcs because we needed these values to identify the leaving arc. For the shortest path problem, however, we can determine the leaving arc without considering the flow values. If (k, l) is the entering arc, then $(\text{pred}(l), l)$ is the leaving arc. Thus the network simplex algorithm for the shortest path problem need not maintain arc flows. Moreover, updating of the tree indices is simpler for the shortest path problem.

The network simplex algorithm for the shortest path problem is similar to the label-correcting algorithms discussed in Section 5.3. Recall that a label-correcting algorithm maintains distance labels $d(i)$, searches for an arc satisfying the condition $d(j) > d(i) + c_{ij}$, and sets the distance label of node j equal to $d(i) + c_{ij}$. In the

network simplex algorithm, if we define $d(i) = -\pi(i)$, then $d(i)$ are the valid distance labels (i.e., they represent the length of some directed path from source to node i). At each iteration the network simplex algorithm selects an arc (i, j) with $c_{ij}^{\pi} < 0$. Observe that $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j) = c_{ij} + d(i) - d(j)$. Therefore, like a label-correcting algorithm, the network simplex algorithm selects an arc that satisfies the condition $d(j) > d(i) + c_{ij}$. The algorithm then increases the potential of every node in the subtree rooted at node j by an amount $| c_{ij}^{\pi} |$ which amounts to decreasing the distance label of all the nodes in the subtree rooted at node j by an amount $| c_{ij}^{\pi} |$. In this regard the network simplex algorithm differs from the label correcting algorithm: instead of updating one distance label at each step, it updates several of them.

If the network contains no negative cycle, the network simplex algorithm would terminate with a shortest path tree. When the network does contain a negative cycle, the algorithm would eventually encounter a situation like that depicted in Figure 11.17. This type of situation will occur only when the tail of the entering arc (k, l) belongs to $D(l)$, the set of descendants of node l. The network simplex algorithm can detect this situation easily without any significant increase in its computational effort: After introducing an arc (k, l), the algorithm updates the potentials of all nodes in $D(l)$; at that time, it can check to see whether $k \in D(l)$, and if so, then terminate. In this case, tracing the predecessor indices would yield a negative cycle.

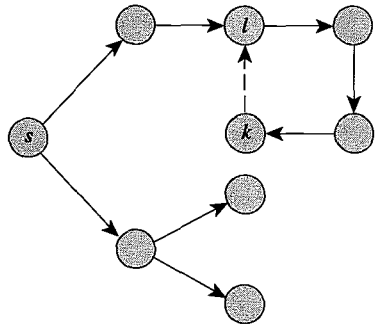


Figure 11.17 Detecting a negative cycle in the network.

The generic version of the network simplex algorithm for the shortest path problem runs in pseudopolynomial time. This result follows from the facts that (1) for each node i, $-nC \leq \pi(i) \leq nC$ (because the length of every directed path from s to node i lies between $-nC$ to nC), and (2) each iteration increases the value of at least one node potential. We can, however, develop special implementations that run in polynomial time. In the remainder of this section, in the exercises, and in the reference notes at the end of this chapter, we describe several polynomial-time implementations of the network simplex algorithm for the shortest path problem. These algorithms will solve the shortest path problem in $O(n^2m)$, $O(n^3)$, and $O(nm \log C)$ time. We obtain these polynomial-time algorithms by carefully selecting the entering arcs.

First eligible arc pivot rule. We have described this pivot rule in Section 11.5. The network simplex algorithm with this pivot rule bears a strong resemblance with the FIFO label-correcting algorithm that we described in Section 5.4. Recall

that the FIFO label-correcting algorithm examines the arc list in a wraparound fashion: If an arc (i, j) violates its optimality condition (i.e., it is eligible), the algorithm updates the distance label of node j. This order for examining the arcs ensures that after the kth pass, the algorithm has computed the shortest path distances to all those nodes that have a shortest path with k or fewer arcs. The network simplex algorithm with the first eligible arc pivot rule also examines the arc list in a wraparound fashion, and if an arc (i, j) is eligible (i.e., violates its optimality condition), it updates the distances label of every node in $D(j)$, which also includes j. Consequently, this pivot rule will also, within k passes, determine shortest path distances to all those nodes that are connected to the source node s by a shortest path with k or fewer arcs. Consequently, the network simplex algorithm will perform at most n passes over the arc list. As a result, the algorithm will perform at most nm pivots and run in $O(n^2m)$ time. In Exercise 11.30 we discuss a modification of this algorithm that runs in $O(n^3)$ time.

Dantzig's pivot rule. This pivot rule selects the entering arc as an arc with the maximum violation. Let C denote the largest arc cost. We will show that the network simplex algorithm with this pivot rule performs $O(n^2 \log(nC))$ pivots and so runs in $O(n^2m \log(nC))$ time.

Scaled pivot rule. This pivot is a scaled variant of Dantzig's pivot rule. In this pivot rule we perform a number of scaling phases with varying values of a scaling parameter Δ. Initially, we let $\Delta = 2^{\lceil \log C \rceil}$ (i.e., we set Δ equal to the smallest power of 2 greater than or equal to C) and pivot in any nontree arc with a violation of at least $\Delta/2$. When no arc has a violation of at least $\Delta/2$, we replace Δ by $\Delta/2$ and repeat the steps. We terminate the algorithm when $\Delta < 1$.

We now show that the network simplex algorithm with the scaled pivot rule solves the shortest path problem in polynomial time. It is easy to verify that Dantzig's pivot rule is a special case of scaled pivot rule, so this result also shows that when implemented with Dantzig's pivot rule, the network simplex algorithm requires polynomial time.

We call the sequence of iterations for which Δ remains unchanged as the *Δ-scaling phase*. Let π denote the set of node potentials at the beginning of a Δ-scaling phase. Moreover, let $P^*(p)$ denote a shortest path from node s to node p and let $\pi^*(p) = -\sum_{(i,j)\in P^*} c_{ij}$ denote the optimal node potential of node p. Our analysis of the scaled pivot rule uses the following lemma:

Lemma 11.8. *If π denotes the current node potentials at the beginning of the Δ-scaling phase, then $\pi^*(p) - \pi(p) \leq 2n\Delta$ for each node p.*

Proof. In the first scaling phase, $\Delta \geq C$ and the lemma follows from the facts that $-nC$ and nC are the lower and upper bounds on any node potentials (why?). Consider next any subsequent scaling phase. Property 9.2 implies that

$$\sum_{(i,j)\in P^*(k)} c_{ij}^{\pi} = \sum_{(i,j)\in P^*(k)} c_{ij} - \pi(s) + \pi(p) = \pi(p) - \pi^*(p). \qquad (11.4)$$

Since Δ is an upper bound on the maximum arc violation at the beginning of the Δ-scaling phase (except the first one), $c_{ij}^{\pi} \geq -\Delta$ for every arc $(i, j) \in A$. Sub-

stituting this inequality in (11.4), we obtain

$$\pi(p) - \pi^*(p) \geq -\Delta | P^*(p) | \geq -n\Delta,$$

which implies the conclusion of the lemma.

Now consider the potential function $\Phi = \sum_{p \in N} (\pi^*(p) - \pi(p))$. The preceding lemma shows that at the beginning of each scaling phase, Φ is at most $2n^2\Delta$. Now, recall from our previous discussion in this section that in each iteration, the network simplex algorithm increases at least one node potential by an amount equal to the violation of the entering arc. Since the entering arc has violation at least $\Delta/2$, at least one node potential increases by $\Delta/2$ units, causing Φ to decrease by at least $\Delta/2$ units. Since no node potential ever decreases, the algorithm can perform at most $4n^2$ iterations in this scaling phase. So, after at most $4n^2$ iterations, either the algorithm will obtain an optimal solution or will complete the scaling phase. Since the algorithm performs $O(\log C)$ scaling phases, it will perform $O(n^2 \log C)$ iterations and so require $O(n^2 m \log C)$ time. It is, however, possible to implement this algorithm in $O(nm \log C)$ time; the reference notes provide a reference for this result.

11.8 NETWORK SIMPLEX ALGORITHM FOR THE MAXIMUM FLOW PROBLEM

In this section we describe another specialization of the network simplex algorithm: its implementation for solving the maximum flow problem. The resulting algorithm is essentially an augmenting path algorithm, so it provides an alternative proof of the max-flow min-cut theorem we discussed in Section 6.5.

As we have noted before, we can view the maximum flow problem as a particular version of the minimum cost flow problem, obtained by introducing an additional arc (t, s) with cost coefficient $c_{ts} = -1$ and an upper bound $u_{ts} = \infty$, and by setting $c_{ij} = 0$ for all the original arcs (i, j) in A. To simplify our notation, we henceforth assume that A represents the set $A \cup \{(t, s)\}$. The resulting formulation is to

$$\text{Minimize} \quad -x_{ts}$$

subject to

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

$$0 \leq x_{ij} \leq u_{ij} \qquad \text{for all } (i, j) \in A.$$

Observe that minimizing $-x_{ts}$ is equivalent to maximizing x_{ts}, which is equivalent to maximizing the net flow sent from the source to the sink, since this flow returns to the source via arc (t, s). This observation explains why the inflow equals the outflow for every node in the network, including the source and the sink nodes.

Note that in any feasible spanning tree solution that carries a positive amount of flow from the source to the sink (i.e., $x_{ts} > 0$), arc (t, s) must be in the spanning tree. Consequently, the spanning tree for the maximum flow problem consists of

two subtrees of G joined by the arc (t, s) (see Figure 11.18). Let T_s and T_t denote the subtrees containing nodes s and t.

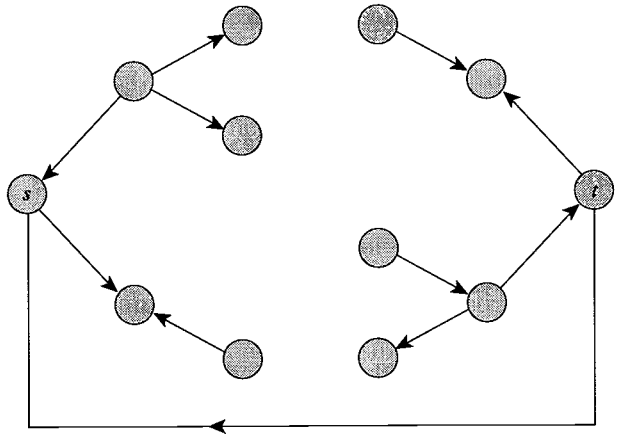


Figure 11.18 Spanning tree for the maximum flow problem.

We obtain node potentials corresponding to a feasible spanning tree of the maximum flow problem as follows. Since we can set one node potential arbitrarily, let $\pi(t) = 0$. Furthermore, since the reduced cost of arc (t, s) must be zero, $0 = c_{ts}^{\pi} = c_{ts} - \pi(t) + \pi(s) = -1 + \pi(s)$, which implies that $\pi(s) = 1$. Since (1) the reduced cost of every arc in T_s and T_t must be zero, and (2) the costs of these arcs are also zero, the node potentials have the following values: $\pi(i) = 1$ for every node $i \in T_s$, and $\pi(i) = 0$ for every node $i \in T_t$.

Notice that every spanning tree solution of the maximum flow problem defines an s–t cut $[S, \bar{S}]$ in the original network obtained by setting $S = T_s$ and $\bar{S} = T_t$. Each arc in this cut is a nontree arc; its flow has value zero or equals the arc's capacity. For every forward arc (i, j) in the cut, $c_{ij}^{\pi} = -1$, and for every backward arc (i, j) in the cut, $c_{ij}^{\pi} = 1$. Moreover, for every arc (i, j) not in the cut, $c_{ij}^{\pi} = 0$. Consequently, if every forward arc in the cut has a flow value equal to the arc's capacity and every backward arc has zero flow, this spanning tree solution satisfies the optimality conditions (11.1), and therefore it must be optimal. On the other hand, if in the current spanning tree solution, some forward arc in the cut has a flow of value zero or the flow on some backward arc equals the arc's capacity, all these arcs have a violation of 1 unit. Therefore, we can select any of these arcs to enter the spanning tree. Suppose that we select arc (k, l). Introducing this arc into the tree creates a cycle that contains arc (t, s) as a forward arc (see Figure 11.19). The algorithm augments the maximum possible flow in this cycle and identifies a blocking arc. Dropping this arc again creates two subtrees joined by the arc (t, s). This new tree constitutes a spanning tree for the next iteration.

Notice that this algorithm is an augmenting path algorithm: The tree structure permits us to determine the path from the source to the sink very easily. In this sense the network simplex algorithm has an advantage over other types of augmenting path algorithms for the maximum flow problem. As a compensating factor, however, due to degeneracy, the network simplex algorithm might not send a positive amount of flow from the source to the sink in every iteration.

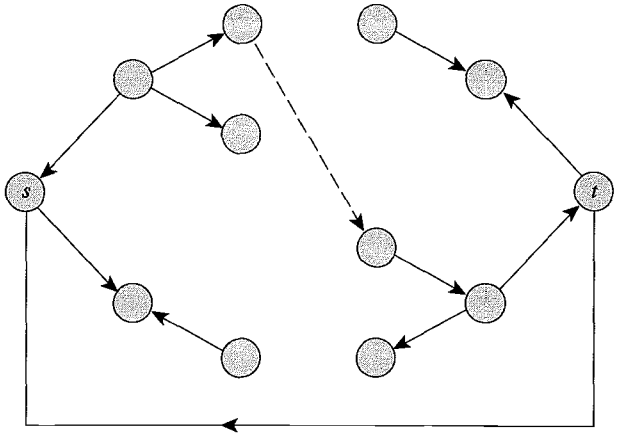


Figure 11.19 Forming a cycle.

The network simplex algorithm for the maximum flow problem gives another proof of the max-flow min-cut theorem. The algorithm terminates when every forward arc in the cut is capacitated and every backward arc has a flow of value zero. This termination condition implies that the current maximum flow value equals the capacity of the s–t cut defined by the subtrees T_s and T_t, and thus the value of a maximum flow from node s to node t equals the capacity of the minimum s–t cut.

Just as the mechanics of the network simplex algorithm becomes simpler in the context of the maximum flow problem, so does the concept of a strongly feasible spanning tree. If we designate the sink as the root node, the definition of a strongly feasible spanning tree implies that we can send a positive amount of flow from every node in T_t to the sink node t without violating any of the flow bounds. Therefore, every arc in T_t whose flow value is zero must point toward the sink node t and every arc in T_t whose flow value equals the arc's upper bound must point away from node t. Moreover, the leaving arc criterion reduces to selecting a blocking arc in the pivot cycle that is farthest from the sink node when we traverse the cycle in the direction of arc (t, s) starting from node t. Each degenerate pivot selects an entering arc that is incident to some node in T_t. The preceding observation implies that each blocking arc must be an arc in T_s. Consequently, each degenerate pivot increases the size of T_t, so the algorithm can perform at most n consecutive degenerate pivots. We might note that the minimum cost flow problem does not satisfy this property: For the more general problem, the number of consecutive degenerate pivots can be exponentially large.

The preceding discussion shows that when implemented to maintain a strongly feasible spanning tree, the network simplex algorithm performs $O(n^2U)$ iterations for the maximum flow problem. This result follows from the fact that the number of nondegenerate pivots is at most nU, an upper bound on the maximum flow value. This bound on the number of iterations is nonpolynomial, so is not satisfactory from a theoretical perspective. Developing a polynomial-time network simplex algorithm for the maximum flow problem remained an open problem for quite some time. However, researchers have recently suggested an entering arc rule that performs only $O(nm)$ iterations and can be implemented to run in $O(n^2m)$ time. This rule

selects entering arcs so that the algorithm augments flow along shortest paths from the source to the sink. We provide a reference for this result in the reference notes.

11.9 RELATED NETWORK SIMPLEX ALGORITHMS

In this section we study two additional algorithms for the minimum cost flow problem—the *parametric network simplex algorithm* and the *dual network simplex algorithm*—that are close relatives of the network simplex algorithm. In contrast to the network simplex algorithm, which maintains a feasible solution at each intermediate step, both of these algorithms maintain an infeasible solution that satisfies the optimality conditions; they iteratively attempt to transform this solution into a feasible solution. The solution maintained by the parametric network simplex algorithm satisfies all of the problem constraints except the mass balance constraints at two specially designated nodes, s and t. The solution maintained by the dual network simplex algorithm satisfies all of the mass balance constraints but might violate the lower and upper bound constraints on some arc flows. Like the network simplex algorithm, both algorithms maintain a spanning tree at every step and perform all computations using the spanning tree.

Parametric Network Simplex Algorithm

For the sake of simplicity, we assume that the network has one supply node (the source s) and one demand node (the sink t). We incur no loss of generality in imposing this assumption because we can always transform a network with several supply and demand nodes into one with a single supply and a single demand node. The parametric network simplex algorithm starts with a solution for which $b'(s) = -b'(t) = 0$, and gradually increases $b'(s)$ and $-b'(t)$ until $b'(s) = b(s)$ and $b'(t) = b(t)$. Let $\mathbf{T}$ be a shortest path tree rooted at the source node s in the underlying network. The parametric network simplex algorithm starts with zero flow and with $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ with $\mathbf{L} = A - \mathbf{T}$ and $\mathbf{U} = \emptyset$ as the initial spanning tree structure. Since, by Assumption 9.5, all the arc costs are nonnegative, the zero flow is an optimal flow provided that $b(s) = b(t) = 0$. Moreover, since $\mathbf{T}$ is a shortest path tree, the shortest path distances $d(\cdot)$ to the nodes satisfy the condition $d(j) = d(i) + c_{ij}$ for each $(i, j) \in \mathbf{T}$, and $d(j) \leq d(i) + c_{ij}$ for each $(i, j) \notin \mathbf{T}$. By setting $\pi(j) = -d(j)$ for each node j, these shortest path optimality conditions become the optimality conditions (11.1) of the initial spanning tree structure $(\mathbf{T}, \mathbf{L}, \mathbf{U})$.

Thus the parametric network simplex algorithm starts with an optimal solution of a minimum cost flow problem that violates the mass balance constraints only at the source and sink nodes. In the subsequent steps, the algorithm maintains optimality of the solution and attempts to satisfy the violated constraints by sending flow from node s to node t along tree arcs. The algorithm stops when it has sent the desired amount $(b(s) = -b(t))$ of flow.

In each iteration the algorithm performs the following computations. Let $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ be the spanning tree structure at some iteration. The spanning tree $\mathbf{T}$ contains a unique path P from node s to node t. The algorithm first determines the maximum amount of flow δ that can be sent from s to t along P while honoring the flow bounds

on the arcs. Let $\overline{P}$ and $\underline{P}$ denote the sets of forward and backward arcs in P. Then

$$\delta = \min[\min\{u_{ij} - x_{ij}:(i, j) \in \overline{P}\}, \min\{x_{ij}:(i, j) \in \underline{P}\}].$$

The algorithm either sends δ units of flow along P, or a smaller amount if it would be sufficient to satisfy the mass balance constraints at nodes s and t. As in the network simplex algorithm, all the tree arcs have zero reduced costs; therefore, sending additional flow along the tree path from node s to node t preserves the optimality of the solution. If the solution becomes feasible after the augmentation, the algorithm terminates. If the solution is still infeasible, the augmentation creates at least one *blocking arc* (i.e., an arc that prohibits us from sending additional flow from node s to node t). We select one such blocking arc, say (p, q), as the *leaving arc* and replace it by some nontree arc (k, l), called the *entering arc*, so that the next spanning tree both (1) satisfies the optimality condition, and (2) permits additional flow to be sent from node s to node t. We accomplish this transition from one tree to another by performing a *dual pivot*. Recall from Section 11.5 that a (primal) pivot first identifies an entering arc and then the leaving arc. In contrast, a dual pivot first selects the leaving arc and then identifies the entering arc.

We perform a dual pivot in the following manner. We first drop the leaving arc from the spanning tree. Doing so gives us two subtrees T_s and T_t, with $s \in T_s$ and $t \in T_t$. Let S and $\overline{S}$ be the subsets of nodes spanned by these two subtrees. Clearly, the cut $[S, \overline{S}]$ is an s–t cut and the entering arc (k, l) must belong to $[S, \overline{S}]$ if the next solution is to be a spanning tree solution. As earlier, we let $(S, \overline{S})$ denote the set of forward arcs and $(\overline{S}, S)$ the set of backward arcs in the cut $[S, \overline{S}]$. Each arc in the cut $[S, \overline{S}]$ is at its lower bound or at its upper bound. We define the set Q of *eligible arcs* as the set

$$Q = ((S, \overline{S}) \cap \mathbf{L}) \cup ((\overline{S}, S) \cap \mathbf{U}),$$

that is, the set of forward arcs at their lower bound and the set of backward arcs at their upper bound. Note that if we add a noneligible arc to the subtrees T_s and T_t, we cannot increase the flow from node s to node t along the new tree path joining these nodes (since the arc lies on the path and would be a forward arc at its upper bound or a backward arc at its lower bound). If we introduce an eligible arc, the new path from node s to node t might be able to carry a positive amount of flow. Next, notice that if $Q = \emptyset$, we can send no additional flow from node s to node t. In fact, the cut $[S, \overline{S}]$ has zero residual capacity and the current flow from node s to node t equals the maximum flow. If $b(s)$ is larger than this flow value, the minimum cost flow problem is infeasible. We now focus on situations in which $Q \neq \emptyset$. Notice that we cannot select an arbitrary eligible arc as the entering arc, because the new spanning tree must also satisfy the optimality condition. For each eligible arc (i, j), we define a number θ_{ij} in the following manner:

$$\theta_{ij} = \begin{cases} c_{ij}^{\pi} & \text{if } (i, j) \in \mathbf{L}, \\ -c_{ij}^{\pi} & \text{if } (i, j) \in \mathbf{U}. \end{cases}$$

Since the spanning tree structure (**T**, **L**, **U**) satisfies the optimality condition (11.1), $\theta_{ij} \geq 0$ for every eligible arc (i, j). Suppose that we select some eligible arc (k, l) as the entering arc. It is easy to see that adding the arc (k, l) to $T_s \cup T_t$ decreases the potential of each node in $\overline{S}$ by θ_{kl} units (throughout the computations, we maintain

that the node potential of the source node s has value zero). This change in node potentials decreases the reduced cost of each arc in $(S, \bar{S})$ by θ_{kl} units and increases the reduced cost of each arc in $(\bar{S}, S)$ by θ_{kl} units. We have four cases to consider.

Case 1. $(i, j) \in (S, \bar{S}) \cap L$

The reduced cost of the arc (i, j) becomes $c_{ij}^{\pi} - \theta_{kl}$. The arc will satisfy the optimality condition (11.1b) if $\theta_{kl} \leq c_{ij}^{\pi} = \theta_{ij}$.

Case 2. $(i, j) \in (S, \bar{S}) \cap U$

The reduced cost of the arc (i, j) becomes $c_{ij}^{\pi} - \theta_{kl}$. The arc will satisfy the optimality condition (11.1c) regardless of the value of θ_{kl} because $c_{ij}^{\pi} \leq 0$.

Case 3. $(i, j) \in (\bar{S}, S) \cap L$

The reduced cost of the arc (i, j) becomes $c_{ij}^{\pi} + \theta_{kl}$. The arc will satisfy the optimality condition (11.1b) regardless of the value of θ_{kl} because $c_{ij}^{\pi} \geq 0$.

Case 4. $(i, j) \in (\bar{S}, S) \cap U$

The reduced cost of the arc (i, j) becomes $c_{ij}^{\pi} + \theta_{kl}$. The arc will satisfy the optimality condition (11.1c) provided that $\theta_{kl} \leq -c_{ij}^{\pi} = \theta_{ij}$.

The preceding discussion implies that the new spanning tree structure will satisfy the optimality conditions provided that

$$\theta_{kl} \leq \theta_{ij} \text{ for each } (i, j) \in ((S, \bar{S}) \cap \mathbf{L}) \cup ((\bar{S}, S) \cap \mathbf{U}) \equiv Q.$$

Consequently, we select the entering arc (k, l) to be an eligible arc for which $\theta_{kl} = \min\{\theta_{ij}:(i, j) \in Q\}$. Adding the arc (k, l) to the subtrees T_s and T_t gives us a new spanning tree structure and completes an iteration. We refer to this dual pivot as *degenerate* if $\theta_{kl} = 0$, and as *nondegenerate* otherwise. We repeat this process until we have sent the desired amount of flow from node s to node t.

It is easy to implement the parametric network simplex algorithm so that it runs in pseudopolynomial time. In this implementation, if an augmentation creates several blocking arcs, we select the one closest to the source as the leaving arc. Using inductive arguments, it is possible to show that in this implementation, the subtree T_s will permit us to augment a positive amount of flow from node s to every other node in T_s along the tree path. Moreover, in each iteration, when the algorithm sends no additional flow from node s to node t, it adds at least one new node to T_s. Consequently, after at most n iterations, the algorithm will send a positive amount of flow from node s to node t. Therefore, the parametric network simplex algorithm will perform $O(nb(s))$ iterations.

To illustrate the parametric network simplex algorithm, let us consider the same example we used to illustrate the network simplex algorithm. Figure 11.20(a) gives the minimum cost flow problem if we choose $s = 1$ and $t = 6$. Figure 11.20(b) shows the tree of shortest paths. All the nontree arcs are at their lower bounds. In the first iteration, the algorithm augments the maximum possible flow from node 1 to node 6 along the tree path 1–2–5–6. This path permits us to send a maximum of $\delta = \min\{u_{12}, u_{25}, u_{56}\} = \min\{8, 2, 6\} = 2$ units of flow. Augmenting 2 units along the path creates the unique blocking arc (2, 5). We drop arc (2, 5) from the tree, creating

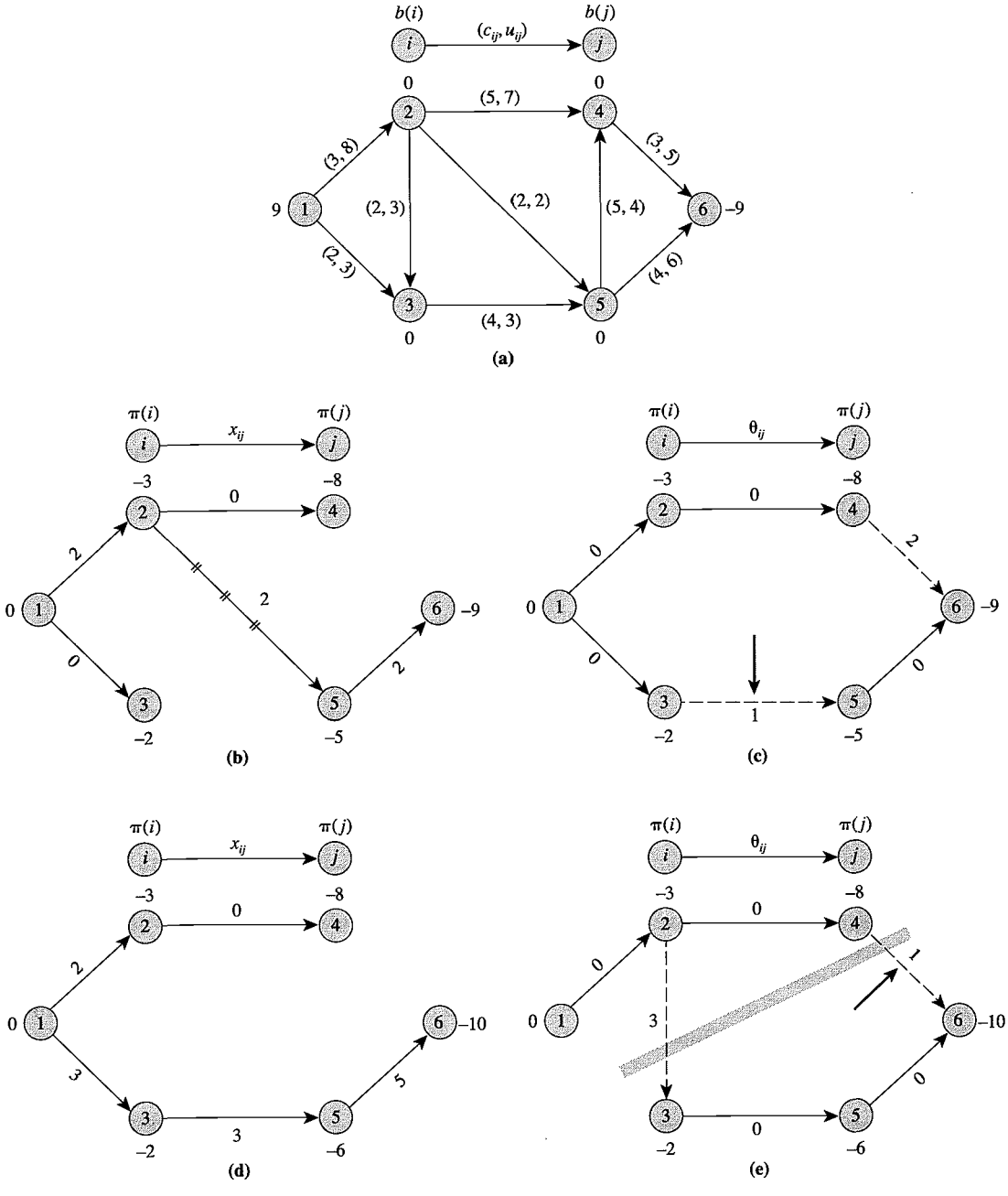


Figure 11.20 Illustrating the parametric network simplex algorithm.

the s–t cut $[S, \bar{S}]$ with $S = \{1, 2, 3, 4\}$ [see Figure 11.20(c)]. This cut contains two eligible arcs: arcs (3, 5) and (4, 6) with $\theta_{35} = 1$ and $\theta_{46} = 2$. We select arc (3, 5) as the entering arc, creating the spanning tree shown in Figure 11.20(d). Notice that the potentials of the nodes 5 and 6 increase by 1 unit. In the new spanning tree, 1–

3–5–6 is the tree path from node 1 to node 6. We augment $\delta = \min\{u_{13}, u_{35}, u_{56} - x_{56}\} = \min\{3, 3, 6 - 2\} = 3$ units of flow along the path, creating two blocking arcs, (1, 3) and (3, 5). The arc (1, 3) is closer to the source and we select it as the leaving arc. As shown in Figure 11.20(e), the resulting s–t cut contains two eligible arcs, (2, 3) and (4, 6). Since $\theta_{46} < \theta_{23}$, we select (4, 6) as the entering arc. We leave the remaining steps of the algorithm as an exercise for the reader.

Notice the resemblance between the parametric network simplex algorithm and the successive shortest path algorithm that we discussed in Section 9.7. Both algorithms maintain the optimality conditions and gradually satisfy the mass balance constraints at the source and sink nodes. Both algorithms send flow along shortest paths from node s to node t. Whereas the successive shortest path algorithm does so by explicitly solving a shortest path problem, the parametric network simplex algorithm implicitly solves a shortest path problem. Indeed, the sequence of iterations that the parametric network simplex algorithm performs between two consecutive positive-flow iterations are essentially the steps of Dijkstra's algorithm for the shortest path problem.

Dual Network Simplex Algorithm

The dual network simplex algorithm maintains a solution that satisfies the mass balance constraints at all nodes, but that violates some of the lower and upper bounds imposed on the arc flows. The algorithm maintains a spanning tree structure (**T**, **L**, **U**) that satisfies the optimality conditions (11.1); the flow on the arcs in **L** and **U** are at their lower and upper bounds, but the flow on the tree arcs might not satisfy their flow bounds. We refer to a tree arc (i, j) as *feasible* if $0 \le x_{ij} \le u_{ij}$ and as *infeasible* otherwise. The algorithm attempts to make infeasible arcs feasible by sending flow along cycles; it terminates when the network contains no infeasible arc.

The dual network simplex algorithm performs a dual pivot at every iteration. Let (**T**, **L**, **U**) be the spanning tree structure at some iteration. In this solution some tree arcs might be infeasible. The algorithm selects any one of these arcs as the leaving arc. (Empirical evidence suggests that choosing an infeasible arc with the maximum violation of its flow bound generally results in a fewer number of iterations.) Suppose that we select the arc (p, q) as the leaving arc and $x_{pq} > u_{pq}$. We later address the case $x_{pq} < 0$. To make the flow on the arc (p, q) feasible, we must decrease the flow on this arc. We decrease the flow by introducing some nontree arc (k, l) that creates a unique cycle W containing arc (p, q) and augment enough flow along this cycle. Let us see which entering arc (k, l) would permit us to accomplish this objective.

If we drop the arc (p, q) from the spanning tree, we create two subtrees T_1 and T_2, with $p \in T_1$ and $q \in T_2$. Let S and $\bar{S}$ be the sets of nodes spanned by T_1 and T_2. In addition, let $(S, \bar{S})$ and $(\bar{S}, S)$ denote the sets of forward and backward arcs in the cut $[S, \bar{S}]$. Each arc in the cut $[S, \bar{S}]$, except the arc (p, q), is at its lower or upper bound. Adding any arc (i, j) in $[S, \bar{S}]$ to **T** creates a unique cycle W that contains the arc (p, q). Suppose that we define the orientation of the cycle W along the arc (i, j) if $(i, j) \in \mathbf{L}$ and opposite to the arc (i, j) if $(i, j) \in \mathbf{U}$. Each nontree arc in the cut $[\bar{S}, S]$ is (1) either a forward arc or a backward arc, and (2) either belongs to **L** or belongs to **U**. Consider any arc $(i, j) \in (S, \bar{S}) \cap \mathbf{L}$. In this case, the orientation

of the cycle is along arc (i, j); consequently, arc (p, q) will be a backward arc in the cycle W and sending additional flow along the orientation of the cycle will decrease flow on the arc (p, q) [see Figure 11.21(a)]. Next, consider any arc $(i, j) \in (\bar{S}, S) \cap \mathbf{U}$. In this case the orientation of the cycle is opposite to arc (i, j); therefore, sending additional flow along the orientation of the cycle again decreases flow on the arc (p, q) [see Figure 11.21(b)]. The reader can easily verify that in the other two cases when $(i, j) \in (S, \bar{S}) \cap \mathbf{U}$ or $(i, j) \in (\bar{S}, S) \cap \mathbf{L}$, increasing flow along the orientation of the cycle does not decrease flow on the arc (p, q). Consequently, we define the set of *eligible arcs* as

$$Q = ((S, \bar{S}) \cap \mathbf{L}) \cup ((\bar{S}, S) \cap \mathbf{U}).$$

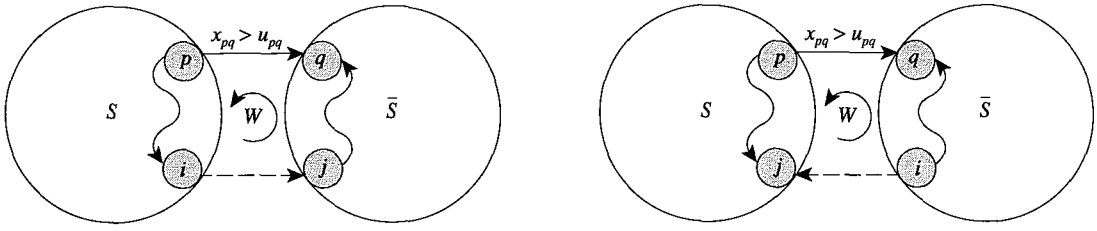


Figure 11.21 Identifying eligible arcs in the dual network simplex algorithm.

If $Q = \varnothing$, we cannot reduce the flow on arc (p, q) and the minimum cost flow problem is infeasible (see Exercise 11.37). If $Q \neq \varnothing$, we must select as the entering arc an eligible arc that would create a new spanning tree structure satisfying the optimality conditions. This step is similar to the same step in the parametric network simplex algorithm. We define a number θ_{ij} for each eligible arc (i, j) in the following manner:

$$\theta_{ij} = \begin{cases} c_{ij}^{\pi} & \text{if } (i, j) \in \mathbf{L}, \\ -c_{ij}^{\pi} & \text{if } (i, j) \in \mathbf{U}, \end{cases}$$

and select an arc (k, l) as the entering arc for which $\theta_{kl} = \min\{\theta_{ij}: (i, j) \in Q\}$. As before, we say this dual pivot is *degenerate* if $\theta_{kl} = 0$ and is *nondegenerate* otherwise. We augment $x_{pq} - u_{pq}$ units of flow along the cycle created by the arc (k, l); doing so decreases the flow on the arc (p, q) to value u_{pq}. Note that as a result of the augmentation, the arc (p, q) becomes feasible; other feasible arcs, however, might become infeasible. In the next spanning tree structure, the arc (k, l) replaces the arc (p, q), and (p, q) becomes a nontree arc at its upper bound. Replacing the arc (p, q) by the arc (k, l) in the spanning tree decreases the potential of each node in $\bar{S}$ by θ_{kl} units. (In the dual network simplex algorithm, the potential of node 1 might not always be zero.) As in our discussion of the parametric network simplex algorithm, it is possible to show that the new spanning tree structure satisfies the optimality conditions.

So far we have addressed situations in which the leaving arc (p, q) is infeasible because $x_{pq} > u_{pq}$. We now consider the case when $x_{pq} < 0$. In this instance, to make this arc feasible, we will increase its flow. The computations in this case are exactly the same as in the previous case except that we define the subtrees T_1 and

T_2 so that $p \in T_2$ and $q \in T_1$. We define the set of eligible arcs as $Q = ((S, \bar{S}) \cap \mathbf{L}) \cup ((\bar{S}, S) \cap \mathbf{U})$ and select an eligible arc (k, l) with the minimum value of θ_{kl} as the entering arc. We augment $| x_{pq} |$ units of flow along the cycle created by the arc (k, l); doing so increases the flow on arc (p, q) to value zero. In the next spanning tree structure, arc (k, l) becomes a tree arc and (p, q) becomes a nontree arc at its lower bound.

Proving the finiteness of the dual network simplex algorithm is easy if each dual pivot is nondegenerate. As before, we assume that $x_{pq} > u_{pq}$ (a similar proof applies when $x_{pq} < 0$). In this case the entering arc (k, l) belongs to $(S, \bar{S}) \cap \mathbf{L}$ or belongs to $(\bar{S}, S) \cap \mathbf{U}$. In the former case, $c^{\pi}_{kl} > 0$ and the flow on the arc (k, l) increases by $(x_{pq} - u_{pq}) > 0$ units. In the latter case, $c^{\pi}_{kl} < 0$ and the flow on the arc decreases by $(x_{pq} - u_{pq}) > 0$ units. In either case, the cost of the flow increases by $c^{\pi}_{kl}(x_{pq} - u_{pq}) > 0$. Since mCU is an upper bound on the objective function value of the minimum cost flow problem and each nondegenerate pivot increases the cost by at least 1 unit, the dual network simplex algorithm will terminate finitely whenever every pivot is nondegenerate. In a degenerate pivot, the objective function value does not change because the entering arc (k, l) satisfies the condition $c^{\pi}_{kl} = 0$. In Exercise 11.38 we describe a dual perturbation technique that avoids the degenerate dual pivots altogether and yields a finite dual network simplex algorithm.

11.10 SENSITIVITY ANALYSIS

The purpose of sensitivity analysis is to determine changes in the optimal solution of the minimum cost flow problem resulting from changes in the data (supply/demand vector, capacity, or cost of any arc). In Section 9.11 we described methods for conducting sensitivity analysis using nonsimplex algorithms. In this section we describe network simplex based algorithms for performing sensitivity analysis.

Sensitivity analysis adopts the following basic approach. We first determine the effect of a given change in the data on the feasibility and optimality of the solution assuming that the spanning tree structure remains unchanged. If the change affects the optimality of the spanning tree structure, we perform (primal) pivots to achieve optimality. Whenever the change destroys the feasibility of the spanning tree structure, we perform dual pivots to achieve feasibility.

Let x^* denote an optimal solution of the minimum cost flow problem. Let ($\mathbf{T}^*$, $\mathbf{L}^*$, $\mathbf{U}^*$) denote the corresponding spanning tree structure and π^* denote the corresponding node potentials. We first consider sensitivity analysis with respect to changes in the cost coefficients.

Cost Sensitivity Analysis

Suppose that the cost of an arc (p, q) increases by λ units. The analysis would be different when arc (p, q) is a tree or a nontree arc.

Case 1. *Arc (p, q) is a nontree arc.*

In this case, changing the cost of arc (p, q) does not change the node potentials of the current spanning tree structure. The modified reduced cost of arc (p, q) is $c^{\pi^*}_{pq} + \lambda$. If the modified reduced cost satisfies condition (11.1b) or (11.1c),

whichever is appropriate, the current spanning tree structure remains optimal. Otherwise, we reoptimize the solution using the network simplex algorithm with (T*, L*, U*) as the starting spanning tree structure.

Case 2. *Arc* (p, q) *is a tree arc.*

In this case, changing the cost of arc (p, q) changes some node potentials. If arc (p, q) is an upward-pointing arc in the current spanning tree, potentials of all the nodes in $D(p)$ increase by λ, and if (p, q) is a downward-pointing arc, potentials of all the nodes in $D(q)$ decrease by λ. Note that these changes alter the reduced costs of those nontree arcs that belong to the cut $[D(q), \overline{D}(q)]$. If all nontree arcs still satisfy the optimality condition, the current spanning tree structure remains optimal; otherwise, we reoptimize the solution using the network simplex algorithm.

Supply/Demand Sensitivity Analysis

To study changes in the supply/demand vector, suppose that the supply/demand $b(k)$ of node k increases by λ and the supply/demand $b(l)$ of another node l decreases by λ. [Recall that since $\sum_{i \in N} b(i) = 0$, the supplies of two nodes must change simultaneously, by equal magnitudes and in opposite directions.] The mass balance constraints require that we must ship λ units of flow from node k to node l. Let P be the unique tree path from node k to node l. Let $\overline{P}$ and $\underline{P}$, respectively, denote the sets of arcs in P that are along and opposite to the direction of the path. The maximum flow change δ_{ij} on an arc $(i, j) \in P$ that preserves the flow bounds is

$$\delta_{ij} = \begin{cases} u_{ij} - x_{ij} & \text{if } (i, j) \in \overline{P}, \\ x_{ij} & \text{if } (i, j) \in \underline{P}. \end{cases}$$

Let

$$\delta = \min\{\delta_{ij} : (i, j) \in P\}.$$

If $\lambda \leq \delta$, we send λ units of flow from node k to node l along the path P. The modified solution is feasible to the modified problem and since the modification in $b(i)$ does not affect the optimality of the solution, the resulting solution must be an optimal solution of the modified problem.

If $\lambda > \delta$, we cannot send λ units of flow from node k to node l along the arcs of the current spanning tree and preserve feasibility. In this case we send δ units of flow along P and reduce λ to $\lambda - \delta$. Let x' denote the updated flow. We next perform a dual pivot (as described in the preceding section) to obtain a new spanning tree that might allow additional flow to be sent from node k to node l along the tree path. In a dual pivot, we first decide on the leaving variable and then identify an entering variable. Let (p, q) be an arc in P that blocks us from sending additional flow from node k to node l. If $(p, q) \in \overline{P}$, then $x'_{pq} = u_{pq}$ and if $(p, q) \in \underline{P}$, then $x'_{pq} = 0$. We drop arc (p, q) from the spanning tree. Doing so partitions the set of nodes into two subtrees. Let S denote the subtree containing node k and $\overline{S}$ denote the subtree containing node l. Now consider the cut $[S, \overline{S}]$. Since we wish to send additional flow through the cut $[S, \overline{S}]$, the arcs eligible to enter the tree would be the forward arcs in the cut at their lower bound or backward arcs at their upper bounds. If the

network contains no eligible arc, we can send no additional flow from node k to node l and the modified problem is infeasible. If the network does contain qualified arcs, then among these arcs, we select an arc, say (g, h), whose reduced cost has the smallest magnitude. We introduce the arc (g, h) into the spanning tree and update the node potentials.

We then again try to send $\lambda' = \lambda - \delta$ units of flow from node k to node l on the tree path. If we succeed, we terminate; otherwise, we send the maximum possible flow and perform another dual pivot to obtain a new spanning tree structure. We repeat these computations until either we establish a feasible flow in the network or discover that the modified problem is infeasible.

Capacity Sensitivity Analysis

Finally, we consider sensitivity analysis with respect to arc capacities. Consider the analysis when the capacity of an arc (p, q) increases by λ units. (Exercise 11.40 considers the situation when an arc capacity decreases by λ units.) Whenever we increase the capacity of any arc, the previous optimal solution always remains feasible; to determine whether this solution remains optimal, we check the optimality conditions (11.1). If arc (p, q) is a tree arc or is a nontree arc at its lower bound, increasing u_{pq} by λ does not affect the optimality condition for that arc. If, however, arc (p, q) is a nontree arc at its upper bound and its capacity increases by λ units, the optimality condition (11.1c) dictates that we must increase the flow on the arc by λ units. Doing so creates an excess of λ units at node q and a deficit of λ units at node p. To achieve feasibility, we must send λ units from node q to node p. We accomplish this objective by using the method described earlier in our discussion of supply/demand sensitivity analysis.

11.11 RELATIONSHIP TO SIMPLEX METHOD

So far in this chapter, we have described the network simplex algorithm as a combinatorial algorithm and used combinatorial arguments to show that the algorithm correctly solves the minimum cost flow problem. This development has the advantage of highlighting the inherent combinatorial structure of the minimum cost flow problem and of the network simplex algorithm. The approach has the disadvantage, however, of not placing the network simplex method in the broader context of linear programming. To help to rectify this shortcoming, in this section we offer a linear programming interpretation of the network simplex algorithm. We show that the network simplex algorithm is indeed an adaptation of the well-known simplex method for general linear programs. Because the minimum cost flow problem is a highly structured linear programming problem, when we apply the simplex method to it, the resulting computations become considerably streamlined. In fact, we need not explicitly maintain the matrix representation (known as the simplex tableau) of the linear program and can perform all the computations directly on the network. As we will see, the resulting computations are exactly the same as those performed by the network simplex algorithm. Consequently, the network simplex algorithm is not a new minimum cost flow algorithm; instead, it is a special implementation of the

well-known simplex method that exploits the special structure of the minimum cost flow problem.

Our discussion in this section requires a basic understanding of the simplex method; Appendix C provides a brief review of this method. As we have noted before, the minimum cost flow problem is the following linear program:

$$\text{Minimize} \quad cx$$

subject to

$$\mathcal{N}x = b,$$

$$0 \leq x \leq u.$$

The bounded variable simplex method for linear programming (or, simply, the simplex method) maintains a *basis structure* (**B**, **L**, **U**) at every iteration and moves from one basis structure to another until it obtains an optimal basis structure. The set **B** is the set of basic variables, and the sets **L** and **U** are the nonbasic variables at their lower and upper bounds. Following traditions in linear programming, we also refer to the variables in **B** as a basis. Let $\mathcal{B}$, $\mathcal{L}$, and $\mathcal{U}$ denote the sets of columns in $\mathcal{N}$ corresponding to the variables in **B**, **L**, and **U**. We refer to $\mathcal{B}$ as a *basis matrix*. Our first result is a graph-theoretic characterization of the basis matrix.

Bases and Spanning Trees

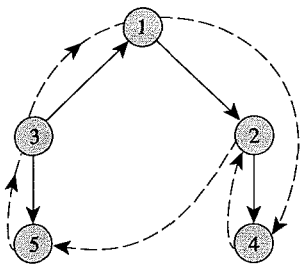
We begin by establishing a one-to-one correspondence between bases of the minimum cost flow problem and spanning trees of G. One implication of this result is that the basis matrix is always lower triangular. The triangularity of the basis matrix is a key in achieving the efficiency of the network simplex algorithm.

We define the jth unit vector e_j as a column vector of size n consisting of all zeros except a 1 in the jth row. We let $\mathcal{N}_{ij}$ denote the column of $\mathcal{N}$ associated with the arc (i, j). In Section 1.2 we show that $\mathcal{N}_{ij} = e_i - e_j$. The rows of $\mathcal{N}$ are linearly dependent since summing all the rows yields the redundant constraint

$$0 = \sum_{i \in N} b(i),$$

which is our assumption that the supplies/demands of all the nodes sum to zero. For convenience we henceforth assume that we have deleted the first row in $\mathcal{N}$ (corresponding to node 1, which is treated as the root node). Thus $\mathcal{N}$ has at most $n - 1$ independent rows. Since the number of linearly independent rows of a matrix is the same as the number of linearly independent columns, $\mathcal{N}$ has at most $n - 1$ linearly independent columns. We show that the $n - 1$ columns associated with arcs of any spanning tree are linearly independent and thus define a basis matrix of the minimum cost flow problem.

Consider a spanning tree **T**. Let $\mathcal{B}$ be the $(n - 1) \times (n - 1)$ matrix defined by the arcs in **T**. As an example, consider the spanning tree shown in Figure 11.22(a) which corresponds to the matrix $\mathcal{B}$ shown in Figure 11.22(b). The first row in this matrix corresponds to the redundant row in $\mathcal{N}$ and deleting this row yields an $(n - 1) \times (n - 1)$ square matrix. For the sake of clarity, however, we shall sometimes retain the first row. We order the rows and columns of $\mathcal{B}$ in a certain specific



(a)

	(1, 2)	(3, 1)	(3, 5)	(2, 4)
1	1	−1	0	0
2	−1	0	0	1
3	0	1	1	0
4	0	0	0	−1
5	0	0	−1	0

(b)

	(2, 4)	(1, 2)	(3, 5)	(3, 1)
4	−1	0	0	0
2	1	−1	0	0
5	0	0	−1	0
3	0	0	1	1
1	0	1	0	−1

(c)

Figure 11.22 (a) Spanning tree and its reverse thread traversal; (b) basis matrix corresponding to the spanning tree; (c) basis matrix after rearranging the rows and columns.

manner. Doing so requires the reverse thread traversal of the nodes in the tree. Recall that a reverse thread traversal visits each node before visiting its predecessor. We order nodes and arcs in the following manner.

1. We order nodes of the tree in order of the reverse thread traversal. For our example, this order is 4–2–5–3–1 [see Figure 11.22(a)].
2. We order the tree arcs by visiting the nodes in order of the reverse thread traversal, and for each node i visited, we select the unique arc incident to it on the path to the root node. For our example, this order is (2, 4), (1, 2), (3, 5), and (3, 1).

We now arrange the rows and columns of $\mathcal{B}$ as specified by the preceding node and arc orderings. Figure 11.22(c) shows the resulting matrix for our example. In this matrix, if we ignore the row corresponding to node 1, we have a lower triangular $(n - 1) \times (n - 1)$ matrix. The triangularity of the matrix is not specific to our example: The matrix would be triangular in general. It is easy to see why. Suppose that the reverse thread traversal selects node i at some step. Let $j = \text{pred}(i)$. Then either $(j, i) \in \mathbf{T}$, or $(i, j) \in \mathbf{T}$. Without any loss of generality, we assume that $(i, j) \in \mathbf{T}$. The reverse thread traversal ensures that we have not visited node j so far. Consequently, the column corresponding to arc (i, j) will contain a $+1$ entry in the row r corresponding to node i, will contain all zero entries above this row, and will contain a -1 entry corresponding to node j below row r (because we will visit node j later). We have thus shown that this rearranged version of $\mathcal{B}$ is a lower triangular matrix and that all of its diagonal elements are $+1$ or -1. We, therefore, have established the following result.

Theorem 11.9 (Triangularity Property). *The rows and columns of the node–arc incidence matrix of any spanning tree can be rearranged to be lower triangular.* ◆

The determinant of a lower triangular matrix is the product of its diagonal elements. Since each diagonal element in the matrix is ± 1, the determinant is ± 1. We now use the well-known fact from linear algebra that a set of $(n - 1)$ column

vectors, each of size $(n - 1)$, is linearly independent if and only if the matrix containing these vectors as columns has a nonzero determinant. This result shows that the columns corresponding to arcs of a spanning tree constitute a basis matrix of $\mathcal{N}$.

We now establish the converse result: Every basis matrix $\mathcal{B}$ of $\mathcal{N}$ defines a spanning tree. The fact that every basis matrix has the same number of columns implies that every basis matrix $\mathcal{B}$ has $(n - 1)$ columns. These columns correspond to a subgraph G' of G having $(n - 1)$ arcs. Suppose that G' contains a cycle W. We assign any orientation to this cycle and consider the expression $\sum_{(i,j)\in W} (\pm 1)\mathcal{N}_{ij} = \sum_{(i,j)\in W} (\pm 1)(e_i - e_j)$; the leading coefficient of each term is $+1$ for those arcs aligned along the orientation of the cycle and is -1 for arcs aligned opposite to the orientation of the cycle. It is easy to verify that for each node j contained in the cycle, the unit vector e_j appears twice, once with a $+1$ sign and once with a -1 sign. Consequently, the preceding expression sums to zero, indicating that the columns corresponding to arcs of a cycle are linearly dependent. Since the columns of $\mathcal{B}$ are linearly independent, G' must be an acyclic graph. Any acyclic graph on n nodes containing $(n - 1)$ arcs must be a spanning tree. So we have established the following theorem.

Theorem 11.10 (Basis Property). *Every spanning tree of G defines a basis of the minimum cost flow problem and, conversely, every basis of the minimum cost flow problem defines a spanning tree of G.* ◆

Implications of Triangularity

In the preceding discussion we showed that we can arrange every basis matrix of the minimum cost flow problem so that it is lower triangular and has an associated spanning tree. We now show that the triangularity of the basis matrix allows us to simplify the computations of the simplex method when applied to the minimum cost flow problem.

When applied to the minimum cost flow problem, the simplex method maintains a basis structure (**B**, **L**, **U**) at every step. Our preceding discussion implies that the arcs in the set **B** constitute a spanning tree and the arcs in the set $\mathbf{L} \cup \mathbf{U}$ are nontree arcs. Therefore, this basis structure is no different from the spanning tree structure that the network simplex algorithm maintains. Moreover, the process of moving from one spanning tree structure to another corresponds to moving from one basis structure to another in the simplex method.

The simplex method performs the following operations:

1. Given a basis structure (**B**, **L**, **U**), determine the associated basic feasible solution.
2. Given a basis structure (**B**, **L**, **U**), determine the associated simplex multipliers π (or, dual variables).
3. Given a basis structure (**B**, **L**, **U**), check whether it is optimal, and if not, then determine an entering nonbasic variable x_{kl}.
4. Given a basis structure (**B**, **L**, **U**) and a nonbasic variable x_{kl}, determine the representation, $\bar{\mathcal{N}}_{kl}$, of the column $\mathcal{N}_{kl}$, corresponding to this variable in terms

of the basis matrix $\mathcal{B}$. We require this representation to perform the pivot operation while introducing the variable x_{kl} into the current basis.

We consider these simplex operations one by one.

Computing the Basic Feasible Solution

Given the basis structure (**B**, **L**, **U**), the simplex method determines the associated basic feasible solution by solving the following system of equations:

$$\mathcal{B}x_{\mathbf{B}} = b - \mathcal{L}x_{\mathbf{L}} - \mathcal{U}x_{\mathbf{U}}. \tag{11.5}$$

In this expression, $x_{\mathbf{B}}$ denotes the set of basic variables, and $x_{\mathbf{L}}$ and $x_{\mathbf{U}}$ denote the sets of nonbasic variables at their lower and upper bounds. The simplex method sets each nonbasic variable in $x_{\mathbf{L}}$ to value zero, each nonbasic variable in $x_{\mathbf{U}}$ to its upper bound, and solves the resulting system of equations. Let $u_{\mathbf{U}}$ be the vector of upper bounds for variables in **U** and let $b' = b - \mathcal{U}u_{\mathbf{U}}$. The simplex method solves the following system of equations:

$$\mathcal{B}x_{\mathbf{B}} = b'. \tag{11.6}$$

Let us see how can we solve (11.6) for the minimum cost flow problem. For simplicity of exposition, assume that $x_{\mathbf{B}} = (x_2, x_3, \ldots, x_n)$. (Assume that the row corresponding to node 1 is the redundant row.) Since $\mathcal{B}$ is a lower triangular matrix, the first row of $\mathcal{B}$ has exactly one nonzero element corresponding to x_2. Therefore, we can uniquely determine the value of x_2. Since the coefficient of x_2 is ± 1, the value of x_2 is integral. The second row of $\mathcal{B}$ has at most two nonzero elements, corresponding to the variables x_2 and x_3. Since we have already determined the value of x_2, we can determine the value of x_3 uniquely. Continuing to solve successively for one variable at a time by this method of forward substitution, we can determine the entire vector $x_{\mathbf{B}}$. Since the nonzero coefficients in the basis matrix $\mathcal{B}$ all have the value ± 1, the only operations we perform are additions and subtractions, which preserve the integrality of the solution.

It is easy to see that the computations required to solve the system of equations $\mathcal{B}x_{\mathbf{B}} = b'$ are exactly same as those performed by the procedure *compute-flows* described in Section 11.4. Recall that the procedure first modifies the supply/demand vector b by setting the flows on the arcs in **U** equal to their upper bounds. The modified supply/demand vector b' equals $b - \mathcal{U}u_{\mathbf{U}}$. Then the procedure examines the nodes in order of the reverse thread traversal and computes the flows on the arcs incident to these nodes. To put the matrix $\mathcal{B}$ into a lower triangular form, we ordered its rows using the reverse thread traversal of the nodes. As a result, the procedure *compute-flows* computes flows on the arcs exactly in the same order as solving the system of equation $\mathcal{B}x_{\mathbf{B}} = b'$ by forward substitution.

Determining the Simplex Multipliers

The simplex algorithm determines the simplex multipliers π associated with a basis structure (**B**, **L**, **U**) by solving the following system of equations:

$$\pi\mathcal{B} = c_{\mathbf{B}}. \tag{11.7}$$

In this expression, $c_{\mathbf{B}}$ is the vector consisting of cost coefficients of the variables in **B**. Assume, for simplicity of exposition, that $\pi = (\pi(2), \pi(3), \ldots, \pi(n))$. Since $\mathcal{B}$ is a lower triangular matrix, the last column of $\mathcal{B}$ has exactly one nonzero element. Therefore, we can immediately determine $\pi(n)$. The second to last column of $\mathcal{B}$ has at most two nonzero elements, corresponding to $\pi(n-1)$ and $\pi(n)$. Since we have already computed $\pi(n)$, we can easily compute $\pi(n-1)$, and so on. We can thus solve (11.7) by backward substitution and compute all the simplex multipliers by performing only additions and subtractions. Since we have arranged the rows of $\mathcal{B}$ in the order of the reverse thread traversal of the nodes, and we determine simplex multipliers in the opposite order, we are, in fact, determining the simplex multipliers of nodes in the order dictated by the thread traversal. Recall from Section 11.4 that the procedure *compute-potentials* also examines nodes and computes the node potentials by visiting the nodes via the thread traversal. Consequently, the procedure *compute-potentials* is in fact solving the system of equations $\pi\mathcal{B} = c_{\mathbf{B}}$ by backward substitution. Also, notice that the node potentials are the simplex multipliers maintained by the simplex method.

Optimality Testing

Given a basis structure (**B**, **L**, **U**), the simplex method computes the simplex multipliers π, and then tests whether the basis structure satisfies the optimality conditions (11.1) (see Appendix C). As expressed in terms of the reduced costs c_{ij}^{π}, the optimality conditions are

$$c_{ij}^{\pi} = c_{ij} - \pi\mathcal{N}_{ij}, \qquad \text{for each } (i, j) \in A.$$

For the minimum cost flow problem, $\mathcal{N}_{ij} = e_i - e_j$ and, therefore, $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$. Consequently, the reduced costs of the arcs as defined in the network simplex algorithm are the linear programming reduced costs and the optimality conditions (11.1) for the network simplex algorithm are the same as the linear programming optimality conditions (see Section C.5). The selection of the entering arc (k, l) in the network simplex algorithm corresponds to selecting the nonbasic variable x_{kl} as the entering variable. To simplify our subsequent exposition, we assume that the entering arc (k, l) is at its lower bound.

Representation of a Nonbasic Column

Once the simplex algorithm has identified a nonbasic variable x_{kl} to enter the basis, it next obtains the representation $\overline{\mathcal{N}}_{kl}$ of the column corresponding to x_{kl} with respect to the current basis matrix. We use this representation to determine the effect on the basic variables of assigning a value θ to x_{kl}, that is, to solve the system

$$x_{\mathbf{B}} = \overline{b}' - \overline{\mathcal{N}}_{kl}\theta.$$

In this expression, $\overline{b}' = \mathcal{B}^{-1}b'$ and $\overline{\mathcal{N}}_{kl} = \mathcal{B}^{-1}\mathcal{N}_{kl}$. Observe that $-\overline{\mathcal{N}}_{kl}$ denotes the change in the values of basic variables as we increase the value of the entering nonbasic variable x_{kl} by 1 unit (i.e., set θ to value 1) and maintain all other nonbasic variables at their current lower and upper bounds. What is the graph-theoretic significance of $\overline{\mathcal{N}}_{kl}$?

The addition of arc (k, l) to the spanning tree T creates exactly one cycle, say W. Define the orientation of the cycle W to align with the orientation of the arc (k, l). Let $\overline{W}$ and $\underline{W}$ denote the sets of forward and backward arcs in W. Observe that if we wish to increase the flow on arc (k, l) by 1 unit, keeping the flow on all other nontree arcs intact, then to satisfy the mass balance constraints we must augment 1 unit of flow along W. This change would increase the flow on arcs in $\overline{W}$ by 1 unit and decrease the flow on arcs in $\underline{W}$ by 1 unit. This discussion shows that the fundamental cycle W created by the nontree arc (k, l) defines the representation $\overline{N}_{kl}$ in the following manner. All the basic variables corresponding to the arcs in $\overline{W}$ have a coefficient of -1 in the column vector $\overline{N}_{kl}$, all the basic variables corresponding to the arcs in $\underline{W}$ have a coefficient of $+1$, and all other basic variables have a coefficient of 0. This discussion also shows that in the network simplex algorithm, augmenting flow in the fundamental cycle created by the entering arc (k, l) and obtaining a new spanning tree solution corresponds to performing a pivot operation and obtaining a new basis structure in the simplex method.

To summarize, we have shown that the network simplex algorithm is the same as the simplex method applied to the minimum cost flow problem. The triangularity of the basis matrix permits us to apply the simplex method directly on the network without explicitly maintaining the simplex tableau. This possibility permits us to use the network structure to greatly improve the efficiency of the simplex method for solving the minimum cost flow problem.

In this section we have shown that the network simplex algorithm is an adaptation of the simplex method for solving general linear programs. A similar development would permit us to show that the parametric network simplex algorithm is an adaptation of the right-hand-side parametric algorithm of linear programming, and that the dual network simplex algorithm is an adaptation of the well-known dual simplex method for solving linear programs. We leave the details of these results as exercises (see Exercises 11.35 and 11.36).

11.12 UNIMODULARITY PROPERTY

In Section 11.4, using network flow algorithms, we established one of the fundamental results of network flows, the integrality property, stating that every minimum cost flow problem with integer supplies/demands and integer capacities has an integer optimal solution. The type of constructive proof that we used to establish this result has the obvious advantage of actually permitting us to compute integer optimal solutions. In that sense, constructive proofs have enormous value. However, constructive proofs do not always identify underlying structural (mathematical) reasons for explaining why results are true. These structural insights usually help in understanding a subject matter, and often suggest relationships between the subject matter and other problem domains or help to define potential limitations and generalization of the subject matter. In this section we briefly examine the structural properties of the integrality property, by providing an algebraic proof of this result. This discussion shows relationships between the integrality property and certain integrality results in linear programming.

Let $\mathcal{A}$ be a $p \times q$ matrix with integer elements and p linearly independent rows (the matrix's rank is p). We say that the matrix $\mathcal{A}$ is *unimodular* if the determinant

of every basis matrix $\mathcal{B}$ of $\mathcal{A}$ has value $+1$ or -1 [i.e., $\det(\mathcal{B}) = \pm 1$]. Recall from Appendix C that a $p \times p$ submatrix of $\mathcal{A}$ is a basis matrix if its columns are linearly independent. The following classical result shows the relationship between unimodularity and the integer solvability of linear programs.

Theorem 11.11 (Unimodularity Theorem). *Let $\mathcal{A}$ be an integer matrix with linearly independent rows. Then the following three conditions are equivalent:*

(a) $\mathcal{A}$ is unimodular.

(b) Every basic feasible solution defined by the constraints $\mathcal{A}x = b, x \geq 0$, is integer for any integer vector b.

(c) Every basis matrix $\mathcal{B}$ of $\mathcal{A}$ has an integer inverse $\mathcal{B}^{-1}$.

Proof. We prove the theorem by showing that $(a) \Rightarrow (b)$, $(b) \Rightarrow (c)$, *and* $(c) \Rightarrow (a)$.

$(a) \Rightarrow (b)$. Each basic feasible solution $x_{\mathbf{B}}$ has an associated basic matrix $\mathcal{B}$ for which $\mathcal{B}x_{\mathbf{B}} = b$. By Cramer's rule, any component x_j of the solution $x_{\mathbf{B}}$ will be of the form

$$x_j = \frac{\det(\text{integer matrix})}{\det(\mathcal{B})}.$$

We obtain the integer matrix in this formula by replacing the jth column of $\mathcal{B}$ with the vector $\mathbf{b}$. Since, by assumption, $\mathcal{A}$ is unimodular, $\det(\mathcal{B})$ is ± 1, so x_j is integer.

$(b) \Rightarrow (c)$. Let $\mathcal{B}$ be a basis matrix of $\mathcal{A}$. Since $\mathcal{B}$ has a nonzero determinant, its inverse $\mathcal{B}^{-1}$ exists. Let e_j denote the jth unit vector (i.e., a vector with a 1 at the jth position and 0 elsewhere). Let $\mathcal{D} = \mathcal{B}^{-1}$ and $\mathcal{D}_j$ denote the jth column of $\mathcal{D}$. We will show that the column vector $\mathcal{D}_j$ is integer for each j whenever condition (b) holds. Select an integer vector α so that $\mathcal{D}_j + \alpha \geq 0$. Let $x = \mathcal{D}_j + \alpha$. Notice that

$$\mathcal{B}x = \mathcal{B}(\mathcal{D}_j + \alpha) = \mathcal{B}(\mathcal{B}^{-1}e_j + \alpha) = e_j + \mathcal{B}\alpha. \tag{11.8}$$

Multiplying the expression (11.8) by $\mathcal{D} = \mathcal{B}^{-1}$, we see that $x = \mathcal{D}_j + \alpha$. Since $e_j + \mathcal{B}\alpha$ is integer (by definition), condition (b) implies that $\mathcal{D}_j + \alpha$ is integer. Recalling that α is integer, we find that $\mathcal{D}_j$ is also integer. This conclusion completes the proof of part (b).

$(c) \Rightarrow (a)$. Let $\mathcal{B}$ be a basis matrix of $\mathcal{A}$. By assumption, $\mathcal{B}$ is an integer matrix, so $\det(\mathcal{B})$ is an integer. By condition (c), $\mathcal{B}^{-1}$ is an integer matrix; consequently, $\det(\mathcal{B}^{-1})$ is also an integer. Since $\mathcal{B} \cdot \mathcal{B}^{-1} = I$ (i.e., an identity matrix), $\det(\mathcal{B}) \cdot \det(\mathcal{B}^{-1}) = 1$, which implies that $\det(\mathcal{B}) = \det(\mathcal{B}^{-1}) = \pm 1$. ◆

This result shows us when a linear program of the form minimize cx, subject to $\mathcal{A}x = b, x \geq 0$, has integer optimal solutions for *all* integer right-hand-side vectors $\mathbf{b}$ and for all cost vectors $\mathbf{c}$. Network flow problems are the largest important class of models that satisfy this integrality property. To establish a formal connection between network flows and the results embodied in this theorem, we consider another noteworthy class of matrices.

Totally unimodular matrices are an important special subclass of unimodular

matrices. We say that a matrix $\mathcal{A}$ is *totally unimodular* if each square submatrix of $\mathcal{A}$ has determinant 0 or ± 1. Every totally unimodular matrix $\mathcal{A}$ is unimodular because each basis matrix $\mathcal{B}$ must have determinant ± 1 (because the zero value of the determinant would imply the linear dependence of the columns of $\mathcal{B}$). However, a unimodular matrix need not be totally unimodular. Totally unimodular matrices are important, in large part, because the constraint matrices of the minimum cost flow problems are totally unimodular.

Theorem 11.12. *The node–arc incidence matrix $\mathcal{N}$ of a directed network is totally unimodular.*

Proof. To prove the theorem, we need to show that every square submatrix $\mathcal{F}$ of $\mathcal{N}$ of size k has determinant 0, $+1$, or -1. We establish this result by performing induction on k. Since each element of $\mathcal{N}$ is 0, $+1$, or -1, the theorem is true for $k = 1$. Now suppose that the theorem holds for some k. Let $\mathcal{F}$ be any $(k + 1) \times (k + 1)$ submatrix of $\mathcal{N}$. The matrix $\mathcal{F}$ satisfies exactly one of the three following possibilities: (1) $\mathcal{F}$ contains a column with no nonzero element; (2) every column of $\mathcal{F}$ has exactly two nonzero elements, in which case, one of these must be a $+1$ and the another a -1; and (3) some column $\mathcal{F}_l$ has exactly one nonzero element, in, say, the ith row. In case (1) the determinant of $\mathcal{F}$ is zero and the theorem holds. In case (2) summing all of the rows in $\mathcal{F}$ yields the zero vector, implying that the rows in $\mathcal{F}$ are linearly dependent and, consequently, $\det(\mathcal{F}) = 0$. In case (3) let $\mathcal{F}'$ denote the submatrix of $\mathcal{F}$ obtained by deleting the ith row and the lth column. Then $\det(\mathcal{F}) = \pm 1 \det(\mathcal{F}')$. By the induction hypothesis, $\det(\mathcal{F}')$ is 0, $+1$, or -1, so $\det(\mathcal{F})$ is also 0, $+1$, or -1. This conclusion establishes the theorem. ◆

This result, combined with Theorem 11.11, provides us with an algebraic proof of the integrality property of network flows: Network flow models have integer optimal solutions because every node–arc incidence matrix is totally unimodular and therefore unimodular. As we will see in later chapters, the constraint matrices for many extensions of the basic network flow problem, for example, generalized flows and multicommodity flows, are not unimodular. Therefore, we would not expect the optimal solutions of these models to be integer even when all of the underlying data are integer. Therefore, to find integer solutions to these problems, we need to rely on methods of integer programming. Although our development of the minimum cost flow problem has not stressed this point, one of the primary reasons that we are able to solve this problem so efficiently, and still obtain integer solutions, is because, as reflected by the integrality property, the basic feasible solutions of the linear programming formulation of this problem are integer whenever the underlying data are integer.

To close this section, we might note that the unimodularity properties provide us with a very strong result: any basic feasible solution is guaranteed to be integer-valued whenever the right-hand-side vector b is integer. It is possible, however, that basic feasible solutions to a linear program might be integer valued for a particular right-hand side even though they might be fractional for some other right-hand sides. We illustrate this possibility in Section 13.8 when we give an integer programming formulation of the minimal spanning tree problem.

11.13 SUMMARY

The network simplex algorithm is one of the most popular algorithms in practice for solving the minimum cost flow problem. This algorithm is an adaptation for the minimum cost flow problem of the well-known simplex method of linear programming. The linear programming basis of the minimum cost flow problem is a spanning tree. This property permits us to simplify the operations of the simplex method because we can perform all of its operations on the network itself, without maintaining the simplex tableau. Our development in this chapter does not require linear programming background because we have developed and proved the validity of the network simplex algorithm from first principles. Later in the chapter we showed the connection between the network simplex algorithm and the linear programming simplex method.

The development in this chapter relies on the fact that the minimum cost flow problem always has an optimal spanning tree solution. This result permits us to restrict our search for an optimal solution among spanning tree solutions. The network simplex algorithm maintains a spanning tree solution and successively transforms it into an improved spanning tree solution until it becomes optimal. At each iteration, the algorithm selects a nontree arc, introduces it into the current spanning tree, augments the maximum possible amount of flow in the resulting cycle, and drops a blocking arc from the spanning tree, yielding a new spanning tree solution. The algorithm is flexible in the sense that we can select the entering arc in a variety of ways and obtain algorithms with different worst-case and empirical attributes.

The network simplex algorithm does not necessarily terminate in a finite number of iterations unless we impose some additional restrictions on the choice of the entering and leaving arcs. We described a special type of spanning tree solution, called the *strongly feasible spanning tree solution*; when implemented in a way that maintains strongly feasible spanning tree solutions, the network simplex algorithm terminates finitely for any choice of the rule used for selecting the entering arc. We can maintain strongly feasible spanning tree solutions by selecting the leaving arc appropriately whenever several arcs qualify to be the leaving arc.

We also specialized the network simplex algorithm for the shortest path and maximum flow problems. When specialized for the shortest path problem, the algorithm maintains a directed out-tree rooted at the source node and iteratively modifies this tree until it becomes a tree of shortest paths. When we specialize the network simplex algorithm for the maximum flow problem, the algorithm maintains an s–t cut and selects an arc in this cut as the entering arc until the associated cut becomes a minimum cut.

The network simplex algorithm has two close relatives that might be quite useful in some circumstances: the parametric network simplex algorithm and the dual network simplex algorithm. The parametric network simplex algorithm maintains a spanning tree solution and parametrically increases the flow from a source node to a sink node until the algorithm has sent the desired amount of flow between these nodes. This algorithm is useful in situations in which we want to maximize the amount of flow to be sent from a source node to a sink node, subject to an upper bound on the cost of flow (see Exercise 10.25). The dual network simplex algorithm maintains a spanning tree solution in which spanning tree arcs do not necessarily satisfy the

flow bound constraints. The algorithm successively attempts to satisfy the flow bound constraints. The primary use of the dual network simplex algorithm has been for reoptimizing the minimum cost flow problem procedures for solving the minimum cost flow problem after we have changed the supply/demand or capacity data.

We also described methods for using the network simplex algorithm to conduct sensitivity analysis for the minimum cost flow problem with respect to the changes in costs, supplies/demands, and capacities. The resulting methods maintain a spanning tree solution and perform primal or dual pivots. Unlike the methods described in Section 9.11, these methods for conducting sensitivity analysis do not necessarily run in polynomial time (without further refinements). However, network simplex-based sensitivity analysis is excellent in practice.

The minimum cost flow problem always has an integer optimal solution; at the beginning of the chapter, we gave an algorithmic proof of this integrality property. We also examined the structural properties of the integrality property by providing an algebraic proof of this result. We showed that the constraint matrix of the minimum cost flow problem is totally unimodular and that, consequently, every basic feasible solution (or, equivalently, spanning tree solution) is an integer solution.

REFERENCE NOTES

Dantzig [1951] developed the network simplex algorithm for the uncapacitated transportation problem by specializing his linear programming simplex method. He proved the spanning tree property of the basis and the integrality property of the optimal solution. Later, his development of the upper bounding technique for linear programming led to an efficient specialization of the simplex method for the minimum cost flow problem. Dantzig's [1962] book discusses these topics.

The network simplex algorithm gained its current popularity in the early 1970s when the research community began to develop and test algorithms using efficient tree indices. Johnson [1966] suggested the first tree indices. Srinivasan and Thompson [1973], and Glover, Karney, Klingman, and Napier [1974] implemented these ideas; these investigations found the network simplex algorithm to be substantially faster than the existing codes that implemented the primal–dual and out-of-kilter algorithms. Subsequent research has focused on designing improved tree indices and determining the best pivot rule. The book by Kennington and Helgason [1980] describes a variety of tree indices and specifies procedures for updating them from iteration to iteration. The book by Bazaraa, Jarvis, and Sherali [1990] also describes a method for updating tree indices. The following papers describe a variety of pivot rules and the computational performance of the resulting algorithms: Glover, Karney, and Klingman [1974], Mulvey [1978], Bradley, Brown, and Graves [1977], Grigoriadis [1986], and Chang and Chen [1989]. The candidate list pivot rule that we describe in Section 11.5 is due to Mulvey [1978]. The reference notes of Chapter 9 contain information concerning the computational performance of the network simplex algorithm and other minimum cost flow algorithms.

Experience with solving large-scale minimum cost flow problems has shown that for certain classes of problems, more than 90% of the pivots in the network simplex algorithm can be degenerate. The strongly feasible spanning tree technique, proposed by Cunningham [1976] for the minimum cost flow problem, and indepen-

dently by Barr, Glover, and Klingman [1977] for the assignment problem, helps to reduce the number of degenerate steps in practice and ensures that the network simplex algorithm has a finite termination. Although the strongly feasible spanning tree technique prevents cycling during a sequence of consecutive degenerate pivots, the number of consecutive degenerate pivots can be exponential. This phenomenon is known as *stalling*. Cunningham [1979] and Goldfarb, Hao, and Kai [1990b] describe several antistalling pivot rules for the network simplex algorithm.

Researchers have attempted, with partial success, to develop polynomial-time implementations of the network simplex algorithm. Tarjan [1991] and Goldfarb and Hao [1988] have described polynomial-time implementations of a variant of the network simplex algorithm that permits pivots to increase value of the objective function. A monotone polynomial-time implementation, in which the value of the objective function is nonincreasing (as it does in any natural implementation), remains elusive to researchers.

Several FORTRAN codes of the network simplex algorithm are available in the public domain. These include (1) the RNET code developed by Grigoriadis and Hsu [1979], (2) the NETFLOW code developed by Kennington and Helgason [1980], and (3) a recent code by Chang and Chen [1989].

We next give selected references for several specific topics.

Shortest path problem. We have adapted the network simplex algorithm for the shortest path problem from Dantzig [1962]. Goldfarb, Hao, and Kai [1990a] and Ahuja and Orlin [1992a] developed the polynomial-time implementations of this algorithm that we have presented in Section 11.7. Additional polynomial-time implementations can be found in Orlin [1985] and Akgül [1985a].

Maximum flow problem. Fulkerson and Dantzig [1955] specialized the network simplex algorithm for the maximum flow problem. Goldfarb and Hao [1990] gave a polynomial-time implementation of this algorithm that performs at most nm pivots and runs in $O(n^2m)$ time; Goldberg, Grigoriadis, and Tarjan [1988] describe an $O(nm \log n)$ implementation of this algorithm.

Assignment problem. One popular implementation of the network simplex algorithm for the assignment problem is due to Barr, Glover, and Klingman [1977]. Roohy-Laleh [1980], Hung [1983], Orlin [1985], Akgül [1985b], and Ahuja and Orlin [1992a] have presented polynomial-time implementations of the network simplex algorithm for the assignment problem. Balinski [1986] and Goldfarb [1985] present polynomial-time dual network simplex algorithms for the assignment problem.

Parametric network simplex algorithm. Schmidt, Jensen, and Barnes [1982], and Ahuja, Batra, and Gupta [1984] are two sources for additional information on the parametric network simplex algorithm.

Dual network simplex algorithm. Ali, Padman, and Thiagarajan [1989] have described implementation details and computational results for the dual network simplex algorithm. Although no one has yet devised a (genuine) polynomial-time primal network simplex algorithm, Orlin [1984] and Plotkin and Tardos [1990]

have developed polynomial-time dual network simplex algorithms. The algorithm of Orlin [1984] is more efficient if capacities satisfy the similarity assumption; otherwise, the algorithm of Plotkin and Tardos [1990] is more efficient. The latter algorithm performs $O(m^2 \log n)$ pivots and runs in $O(m^3 \log n)$ time.

Sensitivity analysis. Srinivasan and Thompson [1972] have described parametric and sensitivity analysis for the transportation problem, which is similar to that for the minimum cost flow problem. Ali, Allen, Barr, and Kennington [1986] also discuss reoptimization procedures for the minimum cost flow problem.

Unimodularity. Hoffman and Kruskal [1956] first proved Theorem 11.11; the proof we have given is due to Veinott and Dantzig [1968]. The book by Schrijver [1986] presents an in-depth treatment of the unimodularity property and related topics.

EXERCISES

11.1. Nurse scheduling problem. A hospital administrator needs to establish a staffing schedule for nurses that will meet the minimum daily requirements shown in Figure 11.23. Nurses reporting to the hospital wards for the first five shifts work for 8 consecutive hours, except nurses reporting for the last shift (2 A.M. to 6 A.M.), when they work for only 4 hours. The administrator wants to determine the minimal number of nurses to employ to ensure that a sufficient number of nurses are available for each period. Formulate this problem as a network flow problem.

Shift	1	2	3	4	5	6
Clock time	6 A.M. to 10 A.M.	10 A.M. to 2 P.M.	2 P.M. to 6 P.M.	6 P.M. to 10 P.M.	10 P.M. to 2 A.M.	2 A.M. to 6 A.M.
Minimum nurses required	70	80	50	60	40	30

Figure 11.23 Nurse scheduling problem.

11.2. Caterer problem. As part of its food service, a caterer needs d_j napkins for each day of the upcoming week. He can buy new napkins at the price of α cents each or have his soiled napkins laundered. Two types of laundry service are available: regular and expedited. The regular laundry service requires two working days and costs β cents per napkin, and the expedited service requires one working day and costs γ cents per napkin ($\gamma > \beta$). The problem is to determine a purchasing and laundry policy that meets the demand at the minimum possible cost. Formulate this problem as a minimum costs flow problem. (*Hint*: Define a network on 15 nodes, 7 nodes corresponding to soiled napkins, 7 nodes corresponding to fresh napkins, and 1 node for the supply of fresh napkins.)

11.3. Project assignment. In a new industry-funded academic program, each master's degree student is required to undertake a 6-month internship project at a company site. Since the projects are such an important component of the student's educational program

and vary considerably by company (e.g., by the problem and industry context) and by geography, each student would like to undertake a project of his or her liking. To assure that the project assignments are "fair," the students and program administrators have decided to use an optimization approach: Each student ranks the available projects in order of increasing preference (lowest to highest). The objective is to assign students to projects to achieve the highest sum of total ranking of assigned projects. The project assignment has several constraints. Each student must work on exactly one project, and each project has an upper limit on the number of students it can accept. Each project must have a supervisor, drawn from a known pool of eligible faculty. Finally, each faculty member has bounds (upper and lower) on the number of projects that he or she can supervise. Formulate this problem as a minimum cost flow problem.

11.4. Passenger routing. United Airlines has six daily flights from Chicago to Washington. From 10 A.M. until 8 P.M., the flights depart every 2 hours. The first three flights have a capacity of 100 passengers and the last three flights can accommodate 150 passengers each. If overbooking results in insufficient room for a passenger on a scheduled flight, United can divert a passenger to a later flight. It compensates any passenger delayed by more than 2 hours from his or her regularly scheduled departure by paying $200 plus $20 for every hour of delay. United can always accommodate passengers delayed beyond the 8 P.M. flight on the 11 P.M. flight of another airline that always has a great deal of spare capacity. Suppose that at the start of a particular day the six United flights have 110, 160, 103, 149, 175, and 140 confirmed reservations. Show how to formulate the problem of determining the most economical passenger routing strategy as a minimum cost flow problem.

11.5. Allocating receivers to transmitters (Dantzig [1962]). An engine testing facility has four types of instruments: α_1 thermocouplers, α_2 pressure gauges, α_3 accelerometers, and α_4 thrust meters. Each instrument measures one type of engine characteristic and transmits its measurements over a separate communication channel. A set of receivers receive and record these data. The testing facility uses four types of receivers, each capable of recording one channel of information: β_1 cameras, β_2 oscilloscopes, β_3 instruments called "Idiots," and β_4 instruments called "Hathaways." The setup time of each receiver depends on the measurement instruments that are transmitting the data; let c_{ij} denote the setup time needed to prepare a receiver of type i to receive the information transmitted from any measurement taken by the jth instrument. The testing facility wants to find an allocation of receivers to transmitters that minimizes the total setup time. Formulate this problem as a network flow problem.

11.6. Faculty–course assignment (Mulvey [1979]). In 1973, the Graduate School of Management at UCLA revamped its M.B.A. curriculum. This change necessitated an increased centralization of the annual scheduling of faculty to courses. The large size of the problem (100 faculty, 500 courses, and three quarters) suggested that a mathematical model would be useful for determining an initial solution. The administration knows the courses to be taught in each of the three teaching quarters (fall, winter, and spring). Some courses can be taught in either of the two specified quarters; this information is available. A faculty member might not be available in all the quarters (due to leaves, sabbaticals, or other special circumstances) and when he is available he might be relieved from teaching some courses by using his project grants for "faculty offset time." Suppose that the administration knows the quarters when a faculty member will be available and the total number of courses he will be teaching in those quarters. The school would like to maximize the preferences of the faculty for teaching the courses. The administration determines these preferences through an annual faculty questionnaire. The preference weights range from -2 to $+2$ and the administration occasionally revises the weights to reflect teaching ability and student inputs. Suggest a network model for determining a teaching schedule.

11.7. Optimal rounding of a matrix (Bacharach [1966], Cox and Ernst [1982]). In Application 6.3 we studied the problem of rounding the entries of a table to their nearest integers

while preserving the row and column sums of the matrix. We refer to any such rounding as a *consistent rounding*. Rounding off an element of the matrix introduces some error. If we round off an element a_{ij} to b_{ij} and $b_{ij} = \lfloor a_{ij} \rfloor$ or $b_{ij} = \lceil a_{ij} \rceil$, we measure the error as $(a_{ij} - b_{ij})^2$. Summing these terms for all the elements of the matrix gives us an error associated with any consistent rounding scheme. We say that a consistent rounding is an *optimal rounding* if the error associated with this rounding is as small as the error associated with any consistent rounding. Show how to determine an optimal rounding by solving a circulation problem. (*Hint*: Construct a network similar to the one used in Application 6.3. Define the arc costs appropriately.)

11.8. Describe an algorithm that either identifies p arc-disjoint directed paths from node s to node t or shows that the network does not contain any such set of paths. In the former case, show how to determine p arc-disjoint paths containing the fewest number of arcs. Suggest modifications of this algorithm to identify p node-disjoint directed paths from node s to node t containing the fewest number of arcs.

11.9. Show that a tree is a directed out-tree T rooted at node s if and only if every node in T except node s has indegree 1. State (but do not prove) an equivalent result for a directed in-tree.

11.10. Suppose that we permute the rows and columns of the node–arc incidence matrix $\mathcal{N}$ of a graph G. Is the modified matrix a node–arc incidence matrix of some graph G'? If so, how are G' and G related?

11.11. Let T be a spanning tree of $G = (N, A)$. Every nontree arc (k, l) has an associated fundamental cycle which is the unique cycle in $T \cup \{(k, l)\}$. With respect to any arbitrary ordering of the arcs $(i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)$, we define the *incidence vector* of any cycle W in G as an m-vector whose kth element is (1) 1, if (i_k, j_k) is a forward arc in W; (2) -1, if (i_k, j_k) is a backward arc in W; and (3) 0, if $(i_k, j_k) \notin W$. Show how to express the incidence vector of any cycle W as a sum of incidence vectors of fundamental cycles.

11.12. Figure 11.24(b) gives a feasible solution of the minimum cost flow problem shown in Figure 11.24(a). Convert this solution into a spanning tree solution with the same or lower cost.

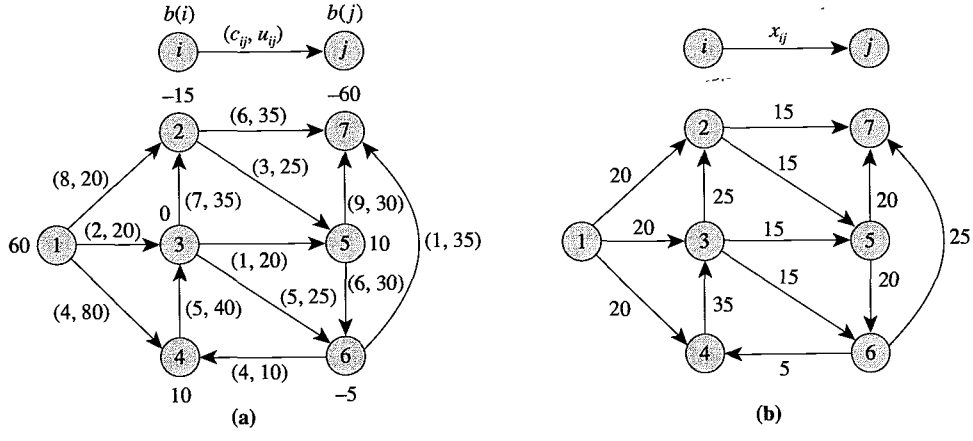


Figure 11.24 Example for Exercise 11.12: (a) problem data; (b) feasible solution.

11.13. Figure 11.25 specifies two spanning trees for the minimum cost flow problem shown in Figure 11.24(a). For Figure 11.25(a), compute the spanning tree solution assuming that all nontree arcs are at their lower bounds. For Figure 11.25(b), compute the spanning tree solution assuming that all nontree arcs are at their upper bounds.

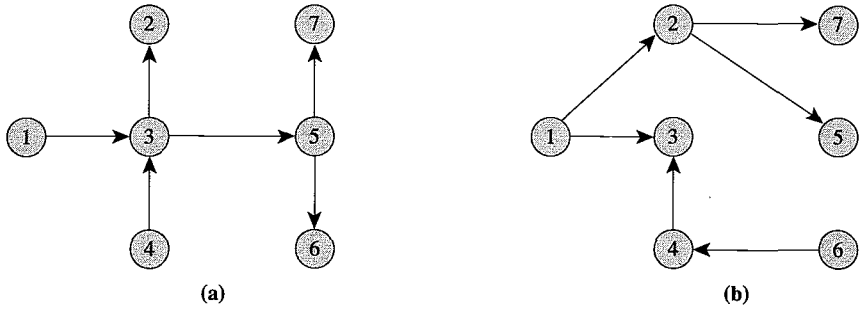


Figure 11.25 Two spanning trees of the network in Figure 11.24.

11.14. Assume that the spanning trees in Figure 11.25 have node 1 as their root. Specify the predecessor, depth, thread, and reverse thread indices of the nodes.

11.15. Compute the node potentials associated with the trees shown in Figure 11.25, which are the spanning trees of the minimum cost flow problem given in Figure 11.24(a). Verify that for each node j, the node potential $\pi(j)$ equals the length of the tree path from node j to the root.

11.16. Consider the minimum cost flow problem shown in Figure 11.26. Using the network simplex algorithm implemented with the first eligible pivot rule, find an optimal solution of this problem. Assume, as always, that arcs are arranged in the increasing order of their tail nodes, and for the same tail node, they are arranged in the increasing order of their head nodes. Use the following initial spanning tree structure: $\mathbf{T} = \{(1, 2), (3, 2), (2, 5), (4, 5), (4, 6)\}$, $\mathbf{L} = \{(3, 5)\}$, and $\mathbf{U} = \{(1, 3), (2, 4), (5, 6)\}$.

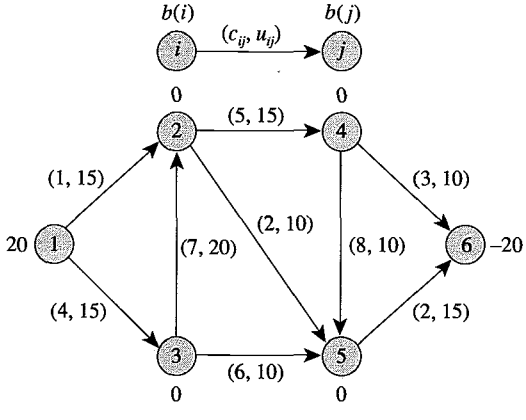


Figure 11.26 Example for Exercises 11.16 and 11.17.

11.17. Using the network simplex algorithm implemented with Dantzig's pivot rule, solve the minimum cost flow problem shown in Figure 11.26. Use the same initial spanning tree structure as used in Exercise 11.16.

11.18. In the procedure *compute-potentials*, we set $\pi(1) = 0$ and then compute other node potentials. Suppose, instead, that we set $\pi(1) = \alpha$ for some $\alpha > 0$ and then recompute all the node potentials. Show that all the node potentials increase by the amount α. Also show that this change does not affect the reduced cost of any arc.

11.19. Justify the procedure *compute-flows* for capacitated networks.

11.20. In the candidate list pivot rule, let *size* denote the maximum allowable size of the candidate list and *iter* denote the maximum number of minor iterations to be performed within a major iteration.

(a) Specify values of size and iter so that the candidate list pivot rule reduces to Dantzig's pivot rule.
(b) Specify values of size and iter so that the candidate list pivot rule reduces to the first eligible arc pivot rule.

11.21. In Section 11.5 we showed how to find the apex of the pivot cycle W in $O(|W|)$ time using the predecessor and depth indices. Show that by using predecessor indices alone, you can find the apex of the pivot cycle in $O(|W|)$ time. (*Hint*: Do so by scanning at most $2|W|$ arcs.)

11.22. Given the predecessor indices of a spanning tree, describe an $O(n)$ time method for computing the thread and depth indices.

11.23. Describe methods for updating the predecessor and depth indices of the nodes when performing a pivot operation. Your method should require $O(n)$ time and should run faster than recomputing these indices from scratch.

11.24. Prove that in a spanning tree we can send a positive amount of flow from any node to the root without violating any flow bound if and only if every tree arc with zero flow is upward pointing and every tree arc at its upper bound is downward pointing.

11.25. Let $G(x)$ denote the residual network corresponding to a flow x. Show that a spanning tree $\mathbf{T}$ is a strongly feasible spanning tree if and only if for every node $i \in N - \{1\}$, $G(x)$ contains the arc $(i, \text{pred}(i))$.

11.26. Primal perturbation. In the minimum cost flow problem on a network G, suppose that we alter the supply/demand vector from value b to value $b + \epsilon$ for some vector ϵ. Let us refer to the modified problem as a *perturbed problem*. We consider the perturbation ϵ defined by $\epsilon(i) = 1/n$ for all $i = 2, 3, \ldots, n$, and $\epsilon(1) = -(n - 1)/n$.
(a) Let $\mathbf{T}$ be a spanning tree of G and let $D(j)$ denote the set of descendants of node j in $\mathbf{T}$. Show that the perturbation decreases the flow on a downward-pointing arc (i, j) by the amount $|D(j)|/n$ and increases the flow on an upward-pointing arc (i, j) by the amount $|D(i)|/n$. Conclude that in a strongly feasible spanning tree solution, each arc flow is nonzero and is an integral multiple of $1/n$.
(b) Use the result in part (a) to show that the network simplex algorithm solves the perturbed problem in pseudopolynomial time irrespective of the pivot rule used for selecting entering arcs.

11.27. Perturbation and strongly feasible solutions. Let $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ be a feasible spanning tree structure of the minimum cost flow problem and let ϵ be a perturbation as defined in Exercise 11.26. Show that $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ is strongly feasible if and only if $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ remains feasible when we replace b by $b + \epsilon$. Use this equivalence to show that when implemented to maintain a strongly feasible basis, the network simplex algorithm runs in pseudopolynomial time irrespective of the pivot rule used for selecting entering arcs.

11.28. Apply the network simplex algorithm to the shortest path problem shown in Figure 11.27(a). Use a depth-first search tree with node 1 as the source node in the initial spanning tree solution and perform three iterations of the algorithm.

11.29. Apply the network simplex algorithm to the maximum flow problem shown in Figure 11.27(b). Use the following spanning tree as the initial spanning tree: a breadth-first search tree rooted at node 1 and spanning the nodes $N - \{t\}$ plus the arc (t, s). Show three iterations of the algorithm.

11.30. Consider the application of the network simplex algorithm, implemented with the following pivot rule, for solving the shortest path problem. We examine all the nodes, one by one, in a wraparound fashion. Each time we examine a node i, we scan all incoming arcs at that node, and if the incoming arcs contain an eligible arc, we pivot in the arc with the maximum violation. We terminate when during an entire pass of the nodes, we find that no arc is eligible. Show when implemented with this pivot rule, the network simplex algorithm would perform $O(n^2)$ pivot operations and would run in $O(n^3)$ time. (*Hint*: The proof is similar to the proof of the first eligible arc pivot rule that we discussed in Section 11.7.)

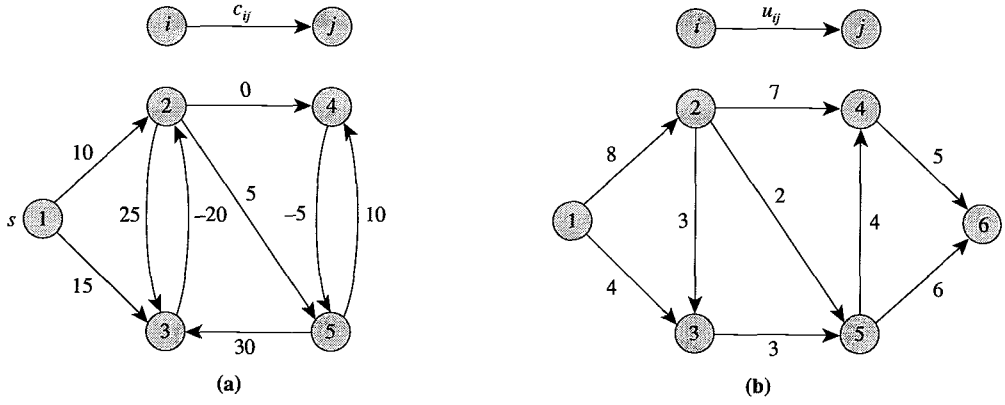


Figure 11.27 Examples for Exercises 11.28 and 11.29.

11.31. The assignment problem, as formulated as a linear programming in (12.1), is a special case of the minimum cost flow problem. Show that every strongly feasible spanning tree of the assignment problem satisfies the following properties: (1) every downward-pointing arc carries unit flow; (2) every upward-pointing arc carries zero flow; and (3) every downward-pointing arc is the unique arc with flow equal to 1 emanating from node i.

11.32. In a strongly feasible spanning tree of the assignment problem, a nontree arc (k, l) is a *downward arc* if node l is a descendant of node k. Show that when the network simplex algorithm, implemented to maintain strongly feasible spanning trees, is applied to the assignment problem, a pivot is nondegenerate if and only if the entering arc is a downward arc.

11.33. Solve the minimum cost flow problem shown in Figure 11.26 by the parametric network simplex algorithm.

11.34. Show how to solve the constrained maximum flow problem, as defined in Exercise 10.25, by a single application of the parametric network simplex algorithm.

11.35. Show that the parametric network simplex algorithm described in Section 11.9 is an adaptation of the right-hand-side parametric simplex method of linear programming. (Consult any linear programming textbook for a review of the parametric simplex method of linear programming.)

11.36. Show that the dual network simplex algorithm described in Section 11.9 is an adaptation of the dual simplex method of linear programming. (Consult any linear programming textbook for a review of the dual simplex method of linear programming).

11.37. At some point during its execution, the dual network simplex algorithm that we discussed in Section 11.9 might find that the set Q of eligible arcs is empty. In this case show that the minimum cost flow problem is infeasible. (*Hint*: Use the result in Exercise 6.43.)

11.38. Dual perturbation. Suppose that we modify the cost vector c of a minimum cost flow problem on a network G in the following manner. After arranging the arcs in some order, we add $\frac{1}{2}$ to the cost of the first arc, $\frac{1}{4}$ to the cost of the second arc, $\frac{1}{8}$ to the cost of the third arc, and so on. We refer to the perturbed cost as c^ϵ, and the minimum cost flow problem with the cost c^ϵ as the *perturbed minimum cost flow problem.*

(a) Show that if x^* is an optimal solution of the perturbed problem, x^* is also an optimal solution of the original problem. (*Hint*: Show that if $G(x^*)$ does not contain any negative cycle with cost c^ϵ, it does not contain any negative cycle with cost c.)

(b) Show that if we apply the dual network simplex algorithm to the perturbed problem, the reduced cost of each nontree arc is nonzero. Conclude that each dual

pivot in the algorithm will be nondegenerate and that the algorithm will terminate finitely. (*Hint*: Use the fact that the reduced cost of a nontree arc (k, l) is the cost of the fundamental cycle created by adding arc (k, l) to the spanning tree.)

11.39. In Exercise 9.24 we considered a numerical example concerning sensitivity analysis of a minimum cost flow problem. Solve the same problem using the simplex-based methods described in Section 11.10.

11.40. In Section 11.10 we described simplex-based procedures for reoptimizing a minimum cost flow solution when some cost coefficient c_{ij} increases or some flow bound u_{ij} decreases. Modify these procedures so that we can use them to handle situations in which (1) some c_{ij} decreases, or (2) some u_{ij} decreases.

11.41. Let $\mathcal{B}$ denote the basis matrix associated with the columns of the spanning tree in Figure 11.25(a). Rearrange the rows and columns of $\mathcal{B}$ so that it is lower triangular.

11.42. Let $G' = (N, A')$ be a subgraph of $G = (N, A)$ containing $| A' | = n - 1$ arcs. Let $\mathcal{B}'$ be the square matrix defined by the columns of arcs in A' (where we delete one redundant row). Show that A' is a spanning tree of G if and only if the determinant of $\mathcal{B}'$ is ± 1.

11.43. Computation of $\mathcal{B}^{-1}$. In this exercise we discuss a combinatorial method for computing the inverse of a basis matrix $\mathcal{B}$ of the minimum cost flow problem. (We assume that we have deleted a redundant row from $\mathcal{B}$.) By definition, $\mathcal{B}\mathcal{B}^{-1} = \mathcal{I}$, an identity matrix. Therefore, the jth column $\mathcal{B}_j^{-1}$ of the inverse matrix $\mathcal{B}^{-1}$ satisfies the condition $\mathcal{B}\mathcal{B}_j^{-1} = e_j$. Consequently, $\mathcal{B}_j^{-1}$ is the unique solution x of the system of equations $\mathcal{B}x = e_j$. Assuming that we have deleted the row corresponding to node 1, x is the flow vector obtained from sending 1 unit of flow from node j to node 1 on the tree arcs corresponding to the basis. Use this result to compute $\mathcal{B}^{-1}$ for the basis $\mathcal{B}$ defined by the spanning trees shown in Figure 11.25(a).

11.44. Show that a matrix $\mathcal{A}$ whose components are 0, +1, or −1 is totally unimodular if it satisfies both of the following conditions: (1) each column of $\mathcal{A}$ contains at most two nonzero elements; and (2) the rows of $\mathcal{A}$ can be partitioned into two subsets $\mathcal{A}_1$ and $\mathcal{A}_2$ so that the two nonzero entries in any column are in the same set of rows if they have different signs and are in different set of rows if they have the same sign.

11.45. Let $\mathcal{N}$ be a totally unimodular matrix. Show that $\mathcal{N}^{\mathbf{T}}$ and $[\mathcal{N}, -\mathcal{N}]$ are also totally unimodular.

11.46. Show that a matrix $\mathcal{N}$ is totally unimodular if and only if the matrix $[\mathcal{N}, \mathcal{I}]$ is unimodular.

11.47. Let $\mathbf{T}$ be a spanning tree of a directed network $G = (N, A)$ with node 1 as a designated root node. Let $d(i, j)$ denote the number of arcs on the tree path from node i to node j in $\mathbf{T}$.

(a) For the given tree $\mathbf{T}$, the *average depth* is $(\sum_{j \in N} d(1, j))/n$, and the *average cycle length* is $(\sum_{\text{nontree arcs } (i,j)} d(i, j) + 1)/(m - n + 1)$. Show that if G is a complete graph, the average cycle length is at most twice the average depth. Show that this relationship is not necessarily valid if the graph is not complete. (*Hint*: Use the fact that the length of the cycle created by adding the arc (i, j) to the tree is at most $d(1, i) + d(1, j) + 1$.)

(b) For a given tree $\mathbf{T}$, let $D(j)$ denote the set of descendants of node j. The *average subtree size* of $\mathbf{T}$ is $(\sum_{j \in N} | D(j) |)/n$. Show that the average subtree size is 1 more than the average depth. (*Hint*: Let $E(j)$ denote the number of ancestors of node j in the tree $\mathbf{T}$. First show that $\sum_{j \in N} | E(j) | = \sum_{j \in N} | D(j) |$.)

11.48. Cost parametrization (Srinivasan and Thompson [1972]). Suppose that we wish to solve a parametric minimum cost flow problem when the cost c_{ij} for each arc $(i, j) \in A$ is given by $c_{ij} = c_{ij}^0 + \lambda c_{ij}^*$ for some constants c_{ij}^0 and c_{ij}^* and we want to find an optimal solution for all values of the parameter λ in a given interval $[\alpha, \beta]$.

(a) Let $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ be an optimal spanning tree structure for the minimum cost flow problem for some value λ of the parameter. Let π^0 denote the node potentials for the tree $\mathbf{T}$ when c_{ij}^0 are the arc costs, and let π^* denote node potentials when

c_{ij}^* are the arc costs in $\mathbf{T}$ (we can compute these potentials using the procedure *compute-potentials*). Show that $\pi^0 + \lambda\pi^*$ are the node potentials for the tree $\mathbf{T}$ when the arc costs are $c_{ij}^0 + \lambda c_{ij}^*$. Use this result to identify the largest value of λ, say $\bar{\lambda}$, for which $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ satisfies the optimality conditions.

(b) Show that at $\lambda = \bar{\lambda}$, some nontree arc (k, l) satisfies its optimality condition as an equality and violates the optimality condition when $\lambda > \bar{\lambda}$. Show that if we perform the pivot operation with arc (k, l) as the entering arc, the new spanning tree structure also satisfies the optimality conditions at $\lambda = \bar{\lambda}$.

(c) Use the results in parts (a) and (b) to solve the minimum cost flow problem for all values of the parameter λ in a given interval $[\alpha, \beta]$.

11.49. Supply/demand parametrization (Srinivasan and Thompson [1972]). Suppose that we wish to solve a parametric minimum cost flow problem in which the supply/demand $b(i)$ of each node $i \in N$ is given by $b(i) = b^0(i) + \lambda b^*(i)$ for some constants $b^0(i)$ and $b^*(i)$ and we want to find an optimal solution for all values of the parameter λ in a given interval $[\alpha, \beta]$. We assume that $\sum_{i \in N} b^0(i) = \sum_{i \in N} b^*(i) = 0$.

(a) Let $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ be an optimal spanning tree structure of the minimum cost flow problem for some value λ of the parameter. Let x_{ij}^0 and x_{ij}^* denote the flows on spanning tree arcs when b^0 and b^* are the supply/demand vectors (we can compute these flows using the procedure *compute-flows*). Show that $x_{ij}^0 + \lambda x_{ij}^*$ is the flow on the spanning tree arcs when $b^0 + \lambda b^*$ is the supply/demand vector. Use this result to identify the largest value of λ, say $\bar{\lambda}$, for which spanning tree arcs satisfy the flow bound constraints.

(b) Show that at $\lambda = \bar{\lambda}$, some tree arc (p, q) satisfies one of its bounds (lower or upper bound) as an equality and violate its flow bound for $\lambda > \bar{\lambda}$. Show that if we perform a dual pivot (as described in Section 11.9) with arc (p, q) as the leaving arc, the new spanning tree structure also satisfies the optimality conditions at $\lambda = \bar{\lambda}$.

(c) Use the results in parts (a) and (b) to solve the minimum cost flow problem for all values of the parameter λ in a given interval $[\alpha, \beta]$.

11.50. Capacity parametrization (Srinivasan and Thompson [1972]). Consider a parametric minimum cost flow problem when the capacity u_{ij} of each arc $(i, j) \in A$ is given by $u_{ij} = u_{ij}^0 + \lambda u_{ij}^*$ for some constants u_{ij}^0 and u_{ij}^*. Describe an algorithm for solving the minimum cost flow problem for all values of the parameter λ in an interval $[\alpha, \beta]$. (*Hint*: Let $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ be the basic structure at some state. Maintain the flow on each arc in the set $\mathbf{U}$ as the arc's upper flow bound (as a function of λ), determine the impact of this choice on the flows on the arcs in the spanning tree, and identify the maximum value of λ for which all the arc flows satisfy their flow bounds.)

11.51. Constrained minimum cost flow problem. The constrained minimum cost flow problem is a minimum cost flow problem with an additional constraint $\sum_{(i,j) \in A} d_{ij} x_{ij} \leq D$, called the *budget constraint*.

(a) Show that the constrained minimum cost flow problem need not satisfy the integrality property (i.e., the problem need not have an integer optimal solution, even when all the data are integer).

(b) For the constrained minimum cost flow problem, we say that a solution x is an *augmented tree solution* if some partition of the arc set A into the subsets $\mathbf{T} \cup \{(p, q)\}$, $\mathbf{L}$, and $\mathbf{U}$ satisfies the following two properties: (1) $\mathbf{T}$ is a spanning tree, and (2) by setting $x_{ij} = 0$ for each arc $(i, j) \in \mathbf{L}$ and $x_{ij} = u_{ij}$ for each arc $(i, j) \in \mathbf{U}$, we obtain a unique flow on the arcs in $\mathbf{T} \cup \{(p, q)\}$ that satisfies the mass balance constraints and the budget constraint. Show that the constrained minimum cost flow problem always has an optimal augmented tree solution. Establish this result in two ways: (1) using a linear programming argument, and (2) using a combinatorial argument like the one we used in proving Theorem 11.2.

REFERENCES

AASHTIANI, H. A., and T. L. MAGNANTI. 1976. Implementing primal–dual network flow algorithms. Technical Report OR 055-76, Operations Research Center, MIT, Cambridge, MA.

ABDALLAOUI, G. 1987. Maintainability of a grade structure as a transportation problem. *Journal of the Operational Research Society* **38,** 367–369.

ADEL'SON-VEL'SKI, G. M., E. A. DINIC, and E. V. KARZANOV. 1975. *Flow Algorithms*. Science, Moscow. (In Russian.)

AHLFELD, D. P., R. S. DEMBO, J. M. MULVEY, and S. A. ZENIOS. 1987. Nonlinear programming on generalized networks. *ACM Transactions on Mathematical Software* **13,** 350–367.

AHO, A. V., J. E. HOPCROFT, and J. D. ULLMAN. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

AHO, A. V., J. E. HOPCROFT, and J. D. ULLMAN. 1983. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

AHUJA, R. K. 1986. Algorithms for the minimax transportation problem. *Naval Research Logistics Quarterly* **33,** 725–740.

AHUJA, R. K., and J. B. ORLIN. 1989. A fast and simple algorithm for the maximum flow problem. *Operations Research* **37,** 748–759.

AHUJA, R. K., and J. B. ORLIN. 1991. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics Quarterly* **38,** 413–430.

AHUJA, R. K., and J. B. ORLIN. 1992a. The scaling network simplex algorithm. *Operations Research* **40,** Supplement 1, S5–S13.

AHUJA, R. K., and J. B. ORLIN. 1992b. Use of representative counts in computational testings of algorithms. Sloan Working Paper, Sloan School of Management, MIT, Cambridge, MA.

AHUJA, R. K., J. L. BATRA, and S. K. GUPTA. 1984. A parametric algorithm for the convex cost network flow and related problems. *European Journal of Operational Research* **16,** 222–235.

AHUJA, R. K., A. V. GOLDBERG, J. B. ORLIN, and R. E. TARJAN. 1992. Finding minimum-cost flows by double scaling. *Mathematical Programming* **53,** 243–266.

AHUJA, R. K., M. KODIALAM, A. K. MISHRA, and J. B. ORLIN. 1992. Computational testing of maximum flow algorithms. Sloan Working Paper, Sloan School of Management, MIT, Cambridge, MA.

AHUJA, R. K., T. L. MAGNANTI, and J. B. ORLIN. 1989. Network flows. In *Handbooks in Operations Research and Management Science*. Vol. 1: *Optimization*, edited by G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd. North-Holland, Amsterdam, pp. 211–369.

AHUJA, R. K., T. L. MAGNANTI, and J. B. ORLIN. 1991. Some recent advances in network flows. *SIAM Review* **33,** 175–219.

AHUJA, R. K., T. L. MAGNANTI, J. B. ORLIN, and M. R. REDDY. 1992. Applications of network optimization. Sloan Working Paper, Sloan School of Management, MIT, Cambridge, MA.

AHUJA, R. K., K. MEHLHORN, J. B. ORLIN, and R. E. TARJAN. 1990. Faster algorithms for the shortest path problem. *Journal of ACM* **37,** 213–223.

AHUJA, R. K., J. B. ORLIN, C. STEIN, and R. E. TARJAN. 1990. Improved algorithms for bipartite network flow problems. Technical Report, Sloan School of Management, MIT, Cambridge, MA. Submitted to *SIAM Journal on Computing*.

AHUJA, R. K., J. B. ORLIN, and R. E. TARJAN. 1989. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing* **18,** 939–954.

AKGÜL, M. 1985a. Shortest path and simplex method. Research Report, Department of Computer Science and Operations Research, North Carolina State University, Raleigh, NC.

AKGÜL, M. 1985b. A genuinely polynomial primal simplex algorithm for the assignment problem. Research Report, Department of Computer Science and Operations Research, North Carolina State University, Raleigh, NC.

ALI, A. I., E. P. ALLEN, R. S. BARR, and J. L. KENNINGTON. 1986. Reoptimization procedures for bounded variable primal simplex network algorithms. *European Journal of Operational Research* **23,** 256–263.

ALI, A. I., D. BARNETT, K. FARHANGIAN, J. L. KENNINGTON, B. PATTY, B. SHETTY, B. MCCARL, and P. WONG. 1984. Multicommodity network problems: Applications and computations. *IIE Transactions* **16,** 127–134.

ALI, A. I., R. V. HELGASON, and J. L. KENNINGTON. 1978. The convex cost network flow problem: A state-of-the-art survey. Technical Report OREM 78001, Southern Methodist University, Dallas, TX.

ALI, A. I., R. PADMAN, and H. THIAGARAJAN. 1989. Dual algorithms for pure network problems. *Operations Research* **37,** 159–171.

ALON, N. 1990. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters* **35,** 201–204.

ANDERSON, W. N. 1975. Maximum matching and the rank of a matrix. *SIAM Journal on Applied Mathematics* **28,** 114–123.

ARISAWA, S., and S. E. ELMAGHRABY. 1977. The "hub" and "wheel" scheduling problems. *Transportation Science* **11,** 124–146.

ARONSON, J. E. 1989. A survey of dynamic network flows. *Annals of Operations Research* **20,** 1–66.

ASSAD, A. A. 1978. Multicommodity network flows: A survey. *Networks* **8,** 37–91.

ASSAD, A. A. 1980a. Models for rail transportation. *Transportation Research* **14A,** 205–220.

ASSAD, A. A. 1980b. Solving linear multicommodity flow problems. *Proceedings of the IEEE International Conference on Circuits and Computers*, pp. 157–161.

BACHARACH, M. 1966. Matrix rounding problems. *Management Science* **9,** 732–742.

BALACHANDRAN, V., and G. L. THOMPSON. 1975. An operator theory of parametric programming for the generalized transportation problems. Parts I–IV. *Naval Research Logistics Quarterly* **22,** 79–125, 297–340.

BALAKRISHNAN, A., T. L. MAGNANTI, and R. T. WONG. 1989. A dual-ascent procedure for large scale uncapacitated network design. *Operations Research* **37,** 716–740.

BALAKRISHNAN, A., T. L. MAGNANTI, A. SHULMAN, and R. T. WONG. 1991. Models for capacity expansion in local access telecommunication networks. *Annals of Operations Research* **33,** 239–284.

BALAKRISHNAN, A., T. L. MAGNANTI, and R. T. WONG. 1991. A decomposition algorithm for local access telecommunications network expansion planning. Working Paper, Operations Research Center, MIT, Cambridge, MA.

BALINSKI, M. L. 1986. A competitive (dual) simplex method for the assignment problem. *Mathematical Programming* **34,** 125–141.

BALL, M. O., and U. DERIGS. 1983. An analysis of alternative strategies for implementing matching algorithms. *Networks* **13,** 517–549.

BARAHONA, F., and É. TARDOS. 1989. Note on Weintraub's minimum cost circulation algorithm. *SIAM Journal on Computing* **18,** 579–583.

BARNHART, C. 1988. A network-based primal–dual solution methodology for the multicommodity network flow problem. Ph.D. dissertation, Department of Civil Engineering, MIT, Cambridge, MA.

BARR, R. S., F. GLOVER, and D. KLINGMAN. 1977. The alternating path basis algorithm for the assignment problem. *Mathematical Programming* **13,** 1–13.

BARR, R. S., and J. S. TURNER. 1981. Microdata file merging through large scale network technology. *Mathematical Programming Study* **15,** 1–22.

BARROS, O., and A. WEINTRAUB. 1986. Spatial market equilibrium problems as network models. *Discrete Applied Mathematics* **13,** 109–130.

BARTHOLDI, J. J., J. B. ORLIN, and H. D. RATLIFF. 1980. Cyclic scheduling via integer programs with circular ones. *Operations Research* **28,** 1074–1085.

BARZILAI, J., W. D. COOK, and M. KRESS. 1986. A generalized network formulation of the pairwise comparison consensus ranking model. *Management Science* **32,** 1007–1014.

BAZARAA, M. S., J. J. JARVIS, and H. D. SHERALI. 1990. *Linear Programming and Network Flows*, 2nd ed. Wiley, New York.

BELFORD, P. C., and H. D. RATLIFF. 1972. A network-flow model for racially balancing schools. *Operations Research* **20,** 619–628.

BELLMAN, R. E. 1957. *Dynamic Programming.* Princeton University Press, Princeton, NJ.

BELLMAN, R. 1958. On a routing problem. *Quarterly of Applied Mathematics* **16,** 87–90.

BELLMORE, M., G. BENNINGTON, and S. LUBORE. 1971. A multivehicle tanker scheduling problem. *Transportation Science* **5,** 36–47.

BENNINGTON, G. E. 1974. Applying network analysis. *Industrial Engineering* **6,** 17–25.

BENTLEY, J. L. 1990. Experiments on geometric traveling salesman heuristics. Computing Science Technical Report 151, AT&T Bell Laboratories, Holmdel, NY.

BENTLEY, J. L., and B. W. KERNIGHAN. 1990. A system for algorithm animation: Tutorial and algorithm animation. Unix Research System Paper, 10th ed., Vol. II. Saunders College Publishing, Philadelphia, pp. 451–475.

BERGE, C. 1957. Two theorems in graph theory. *Proceedings of the National Academy of Sciences USA* **43,** 842–844.

BERGE, C., and A. GHOUILA-HOURI. 1962. *Programming, Games and Transportation Networks.* Wiley, New York.

BERRISFORD, H. G. 1960. The economic distribution of coal supplies in the gas industry: An application of the linear programming transport theory. *Operations Research Quarterly* **11,** 139–150.

BERTSEKAS, D. P. 1976. *Dynamic Programming and Stochastic Control.* Academic Press, New York.

BERTSEKAS, D. P. 1979. A distributed algorithm for the assignment problem. Working Paper, Laboratory for Information and Decision Systems, MIT, Cambridge, MA.

BERTSEKAS, D. P. 1988. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of Operations Research* **14,** 105–123.

BERTSEKAS, D. P., and D. E. BAZ. 1987. Distributed asynchronous relaxation methods for convex network flow problems. *SIAM Journal on Control and Optimization* **25,** 74–85.

BERTSEKAS, D. P., and J. ECKSTEIN. 1988. Dual coordinate step methods for linear network flow problems. *Mathematical Programming B* **42,** 203–243.

BERTSEKAS, D. P., P. A. HOSEIN, and P. TSENG. 1987. Relaxation methods for network flow problems with convex arc costs. *SIAM Journal on Control and Optimization* **25,** 1219–1243.

BERTSEKAS, D. P., and P. TSENG. 1988a. The relax codes for linear minimum cost network flow problems. In *FORTRAN Codes for Network Optimization,* edited by B. Simeone, P. Toth, G. Gallo, F. Maffioli, and S. Pallottino. *Annals of Operations Research* **13,** 125–190.

BERTSEKAS, D. P., and P. TSENG. 1988b. Relaxation methods for minimum cost ordinary and generalized network flow problems. *Operations Research* **36,** 93–114.

BERTSIMAS, D., and J. B. ORLIN. 1991. A technique for speeding up the solution of the Lagrangian dual. Working Paper OR 248-91, Operations Research Center, MIT, Cambridge, MA.

BIXBY, R. E. 1982. Matroids and operations research. In *Advanced Techniques in the Practice of Operations Research,* edited by H. J. Greenberg, F. H. Murphy, and S. H. Shaw. North-Holland, Amsterdam, pp. 433–458.

BIXBY, R. E. 1991. The simplex method: It keeps getting better. Presented at the *14th International Symposium on Mathematical Programming,* Amsterdam, The Netherlands.

BLAND, R. G., and D. L. JENSEN. 1992. On the computational behavior of a polynomial-time network flow algorithm. *Mathematical Programming* **54,** 1–39.

BOAS, P. V. E., R. KAAS, and E. ZIJLSTRA. 1977. Design and implementation of an efficient priority queue. *Mathematical Systems Theory* **10,** 99–127.

BODIN, L. D., B. L. GOLDEN, A. D. SCHUSTER, and W. ROWING. 1980. A model for the blockings of trains. *Transportation Research* **14B,** 115–120.

BODIN, L. D., B. L. GOLDEN, A. A. ASSAD, and M. O. BALL. 1983. Routing and scheduling of vehicles and crews: The state of the art. *Computers and Operations Research* **10,** 69–211.

BONDY, J. A., and U. S. R. MURTY. 1976. *Graph Theory with Applications.* American Elsevier, New York.

BORŮVKA, O. 1926, Příspevěk k řešení otázky ekonomické stavby elektrovodních sítí. *Elektrotechnicky Obzor* **15,** 153–154.

BRADLEY, G., G. BROWN, and G. GRAVES. 1977. Design and implementation of large scale primal transshipment algorithms. *Management Science* **21,** 1–38.

BRADLEY, S. P., A. C. HAX, and T. L. MAGNANTI. 1977. *Applied Mathematical Programming*. Addison-Wesley, Reading, MA.

BROGAN, W. L. 1989. Algorithm for ranked assignments with application to multiobject tracking. *Journal of Guidance*, 357–364.

BROWN, M. H. 1988. *Algorithm Animation*. MIT Press, Cambridge, MA.

BROWN, G. G., and R. D. MCBRIDE. 1984. Solving generalized networks. *Management Science* **30**, 1497–1523.

BRUYNOOGHE, M., A. GIBERT, and M. SAKAROVITCH. 1968. Une méthode d'affection du traffic. In: *Fourth International Symposium on the Theory of Traffic Flow*, Karlsruhe, 1968, W. Lentzback and P. Barons (eds.), Beiträge Theorie des Verkehrsflusses Strassenbau und Strassenkehrstechnik Heft 86, Herausgeben von Bunderesminister fur Verkehr, Abteilung Strassenbau, Bonn, Germany.

BUSAKER, R. G., and P. J. GOWEN. 1961. A procedure for determining minimal-cost network flow patterns. ORO Technical Report 15, Operational Research Office, Johns Hopkins University, Baltimore, MD.

BUSACKER, R. G., and T. L. SAATY. 1965. *Finite Graphs and Networks*. McGraw-Hill, New York.

CABOT, A. V., R. L. FRANCIS, and M. A. STARY. 1970. A network flow solution to a rectilinear distance facility location problem. *AIIE Transactions* **2**, 132–141.

CAHN, A. S. 1948. The warehouse problem (Abstract). *Bulletin of the American Mathematical Society* **54**, 1073.

CARPENTO, G., S. MARTELLO, and P. TOTH. 1988. Algorithms and codes for the assignment problem. In *FORTRAN Codes for Network Optimization*, edited by B. Simeone, P. Toth, G. Gallo, F. Maffioli, and S. Pallottino. *Annals of Operations Research* **13**, 193–224.

CARRARESI, P., and G. GALLO. 1984. Network models for vehicle and crew scheduling. *European Journal of Operational Research* **16**, 139–151.

CHALMET, L. G., R. L. FRANCIS, and P. B. SAUNDERS. 1982. Network models for building evacuation. *Management Science* **28**, 86–105.

CHANDRASEKARAN, R. 1977. Minimum ratio spanning trees. *Networks* **7**, 335–342.

CHANG, M. D., and C. J. CHEN. 1989. An improved primal simplex variant for pure processing networks. *ACM Transactions on Mathematical Software* **15**, 64–78.

CHARNES, A., and D. KLINGMAN. 1971. The "more for less" paradox in the distribution model. *Cahiers du Centre D'Etudes de Recherche Operationnelle* **13**, 11–22.

CHEN, H., and C. G. DEWALD. 1974. A generalized chain labeling algorithm for solving multicommodity flow problems. *Computers and Operations Research* **1**, 437–465.

CHENG, C. K., and T. C. HU. 1990. Ancestor tree for arbitrary multi-terminal cut functions. *Proceedings of a Conference on "Integer Programming and Combinatorial Optimization,"* edited by R. Kannan and W. R. Pulleyblank. University of Waterloo, Waterloo, Canada.

CHERIYAN, J., and T. HAGERUP. 1989. A randomized maximum-flow algorithm. *Proceedings of the 30th IEEE Conference on the Foundations of Computer Science*, pp. 118–123.

CHERIYAN, J., T. HAGERUP, and K. MEHLHORN. 1990. Can a maximum flow be computed in $O(nm)$ time? *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pp. 235–248.

CHERIYAN, J., and S. N. MAHESHWARI. 1989. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing* **18**, 1057–1086.

CHESHIRE, M., K. I. M. MCKINNON, and H. P. WILLIAMS. 1984. The efficient allocation of private contractors to public works. *Journal of the Operational Research Quarterly* **35**, 705–709.

CHIN, F., and D. HOUCH. 1978. Algorithms for updating spanning trees. *Journal of Computer and System Sciences* **16**, 333–344.

CHRISTOPHIDES, N. 1975. *Graph Theory: An Algorithmic Approach*. Academic Press, New York.

CHVÁTAL, V. 1983. *Linear Programming*. W. H. Freeman, New York.

CLARK, J. A., and N. A. J. HASTINGS. 1977. Decision networks. *Operational Research Quarterly* **20**, 51–68.

CLARKE, S., and J. SURKIS. 1968. An operations research approach to racial desegregation of school systems. *Socio-Economic Planning Sciences* **1**, 259–272.

COBHAM, A. 1964. The intrinsic computational difficulty of functions. *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, North-Holland, Amsterdam, pp. 24–30.

COLLINS, M., L. COOPER, R. HELGASON, J. KENNINGTON, and L. LEBLANC. 1978. Solving the pipe network analysis problem using optimization techniques. *Management Science* **24**, 747–760.

COOK, S. 1971. The complexity of theorem proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pp. 151–158.

CORMEN, T. H., C. L. LEISERSON, and R. L. RIVEST. 1990. *Introduction to Algorithms*. MIT Press and McGraw-Hill, New York.

COX, L. H., and L. R. ERNST. 1982. Controlled rounding. *INFOR* **20,** 423–432.

CRAINIC, T., J. A. FERLAND, and J. M. ROUSSEAU. 1984. A tactical planning model for rail freight transportation. *Transportation Science* **18,** 165–184.

CREMEANS, J. E., R. A. SMITH, and G. R. TYNDALL. 1970. Optimal multicommodity network flows with resource allocation. *Naval Research Logistics Quarterly* **17,** 269–280.

CROWDER, H. P., R. S. DEMBO, and J. M. MULVEY. 1978. Reporting computational experiments in mathematical programming. *Mathematical Programming* **15,** 316–329.

CROWDER, H. P., R. S. DEMBO, and J. M. MULVEY. 1979. On reporting computational experiments with mathematical software. *ACM Transactions on Mathematical Software* **5,** 193–203.

CROWDER, H. P., and P. B. SAUNDERS. 1980. Results of a survey on MP performance indicators. *COAL Newsletter*, January, pp. 2–6.

CRUM, R. L., and D. J. NYE. 1981. A network model of insurance company cash flow management. *Mathematical Programming* **15,** 86–101.

CUNNINGHAM, W. H. 1976. A network simplex method. *Mathematical Programming* **11,** 105–116.

CUNNINGHAM, W. H. 1979. Theoretical properties of the network simplex method. *Mathematics of Operations Research* **4,** 196–208.

DAFERMOS, S., and A. NAGURNEY. 1984. A network formulation of market equilibrium problems and variational inequalities. *Operations Research Letters* **5,** 247–250.

DANIEL, R. C. 1973. Phasing out capital equipment. *Operations Research Quarterly* **24,** 113–116.

DANTZIG, G. B. 1951. Application of the simplex method to a transportation problem. In *Activity Analysis and Production and Allocation*, edited by T. C. Koopmans. Wiley, New York, pp. 359–373.

DANTZIG, G. B. 1960. On the shortest route through a network. *Management Science* **6,** 187–190.

DANTZIG, G. B. 1962. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ.

DANTZIG, G. B., W. BLATTNER, and M. R. RAO. 1966. Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem. In *Theory of Graphs. International Symposium*. Dunod, Paris, and Gordon and Breach, New York, pp. 209–213.

DANTZIG, G. B., and D. R. FULKERSON. 1954. Minimizing the number of tankers to meet a fixed schedule. *Naval Research Logistics Quarterly* **1,** 217–222.

DANTZIG, G. B., and P. WOLFE. 1960. Decomposition principle for linear programs. *Operations Research* **8,** 101–111.

DANTZIG, G. B., and P. WOLFE. 1961. The decomposition method for linear programming. *Econometrica* **29,** 767–778.

DEARING, P. M., and R. L. FRANCIS. 1974. A network flow solution to a multifacility minimax location problem involving rectilinear distances. *Transportation Science* **8,** 126–141.

DEMBO, R. S., J. M. MULVEY, and S. A. ZENIOS. 1989. Large-scale nonlinear network models and their applications. *Operations Research* **37,** 353–372.

DENARDO, E. V. 1982. *Dynamic Programming: Models and Applications*. Prentice Hall, Englewood Cliffs, NJ.

DENARDO, E. V., and B. L. FOX. 1979. Shortest-route methods: 1. Reaching, pruning and buckets. *Operations Research* **27,** 161–186.

DENARDO, E. V., U. G. ROTHBLUM, and A. J. SWERSEY. 1988. A transportation problem in which costs depend on the order of arrival. *Management Science* **34,** 774–783.

DEO, N., and C. PANG. 1984. Shortest path algorithms: Taxonomy and annotation. *Networks* **14,** 275–323.

DERIGS, U. 1988. *Programming in Networks and Graphs*. Lecture Notes in Economics and Mathematical Systems, Vol. 300. Springer-Verlag, New York.

DERIGS, U., and W. MEIER. 1989. Implementing Goldberg's max-flow algorithm: A computational investigation. *Zeitschrift für Operations Research* **33,** 383–403.

DERMAN, C., and M. KLEIN. 1959. A note on the optimal depletion of inventory. *Management Science* **5,** 210–214.

DEVINE, M. V. 1973. A model for minimizing the cost of drilling dual completion oil wells. *Management Science* **20,** 532–535.

DEWAR, M. S. J., and H. C. LONGUET-HIGGINS. 1952. The correspondence between the resonance and molecular orbital theories. *Proceedings of the Royal Society of London* **A214,** 482–493.

DIAL, R. 1969. Algorithm 360: Shortest path forest with topological ordering. *Communications of ACM* **12,** 632–633.

DIAL, R., F. GLOVER, D. KARNEY, and D. KLINGMAN. 1979. A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. *Networks* **9,** 215–248.

DIJKSTRA, E. 1959. A note on two problems in connexion with graphs. *Numeriche Mathematics* **1,** 269–271.

DINIC, E. A. 1970. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady* **11,** 1277–1280.

DINIC, E. A. 1973. The method of scaling and transportation problems. *Issled. Diskret. Mat. Science,* Moscow. (In Russian.)

DIRICKX, Y. M. I., and L. P. JENNERGREN. 1975. An analysis of the parking situation in the downtown area of West Berlin. *Transportation Research* **9,** 1–11.

DIVOKY, J. J., and M. S. HUNG. 1990. Performance of shortest path algorithms in network flow problems. *Management Science* **36,** 661–673.

DORSEY, R. C., T. J. HODGSON, and H. D. RATLIFF. 1974. A production scheduling problem with batch processing. *Operations Research* **22,** 1271–1279.

DORSEY, R. C., T. J. HODGSON, and H. D. RATLIFF. 1975. A network approach to a multi-facility, multi-product production scheduling problem without backordering. *Management Science* **21,** 813–822.

DRESS, A. W. M., and T. F. HAVEL. 1988. Shortest path problems and molecular conformation. *Discrete Applied Mathematics* **19,** 129–144.

DROR, M., P. TRUDEAU, and S. P. LADANY. 1988. Network models for seat allocation on flights. *Transportation Research* **22B,** 239–250.

DUDE, R. O., and P. E. HART. 1973. *Pattern Classification and Science Analysis.* Wiley, New York.

EDMONDS, J. 1965a. Paths, trees, and flowers. *Canadian Journal of Mathematics* **17,** 449–467.

EDMONDS, J. 1965b. Maximum matchings and a polyhedran with 0, 1 vertices. *Journal of Research of the National Bureau of Standards* **69B,** 125–130.

EDMONDS, J. 1965c. Minimum partition of a matroid into independent subsets. *Journal of Research of the National Bureau of Standards* **69B,** 67–72.

EDMONDS, J. 1967. An introduction to matching. Mimeographed notes, Engineering Summer Conference, The University of Michigan, Ann Arbor, MI.

EDMONDS, J. 1971. Matroids and the greedy algorithm. *Mathematical Programming* **1,** 127–136.

EDMONDS, J., and E. L. JOHNSON. 1973. Matching, Euler tours and the Chinese postman. *Mathematical Programming* **5,** 88–124.

EDMONDS, J., and R. M. KARP. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM* **19,** 248–264.

ELAM, J., F. GLOVER, and D. KLINGMAN. 1979. A strongly convergent primal simplex algorithm for generalized networks. *Mathematics of Operations Research* **4,** 39–59.

ELIAS, P., A. FEINSTEIN, and C. E. SHANNON. 1956. Note on maximum flow through a network. *IRE Transactions on Information Theory* **IT-2,** 117–119.

ELMAGHRABY, S. E. 1978. *Activity Networks: Project Planning and Control by Network Models.* Wiley–Interscience, New York.

ERLENKOTTER, D. 1978. A dual-based procedure for uncapacitated facility location. *Operations Research* **26,** 992–1009.

ERVOLINA, T. R., and S. T. MCCORMICK. 1990a. Cancelling most helpful cuts for minimum cost network flow. Faculty of Commerce Working Paper 90-MSC-018, University of British Columbia, Vancouver, Canada.

ERVOLINA, T. R., and S. T. MCCORMICK. 1990b. Two strongly polynomial cut cancelling algorithms for minimum cost network flow. Technical Report, Faculty of Commerce and Business Administration, University of British Columbia, Vancouver, Canada.

ESAU, L. R., and K. C. WILLIAMS. 1966. On teleprocessing system design II. *IBM Systems Journal* **5,** 142–147.

EVANS, J. R. 1977. Some network flow models and heuristics for multiproduct production and inventory planning. *AIIE Transactions* **9,** 75–81.

EVANS, J. R. 1984. The factored transportation problem. *Management Science* **30,** 1021–1024.

EVEN, S. 1979. *Graph Algorithms*. Computer Science Press, Rockville, MD.

EVEN, S., and O. KARIV. 1975. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pp. 100–112.

EVEN, S., and R. E. TARJAN. 1975. Network flow and testing graph connectivity. *SIAM Journal on Computing* **4**, 507–518.

EVERETT, H., III. 1963. Generalized Lagrange multiplier method for solving problems of optimal allocation of resources. *Operations Research* **11**, 399–417.

EWASHKO, T. A., and R. C. DUDDING. 1971. Application of Kuhn's Hungarian assignment algorithm to posting servicemen. *Operations Research* **19**, 991.

FARINA, R. F., and F. W. GLOVER. 1983. The application of generalized networks to choice of raw materials for fuels and petrochemicals. In *Energy Models and Studies*, edited by B. Lev. North-Holland, Amsterdam.

FARLEY, A. R. 1980. Levelling terrain trees: A transshipment problem. *Information Processing Letters* **10**, 189–192.

FARVOLDEN, J. M., and W. B. POWELL. 1990. A primal partitioning solution for multicommodity network flow problems. Working Paper 90-04, Department of Industrial Engineering, University of Toronto, Toronto, Canada.

FEDERGRUEN, A., and H. GROENEVELT. 1986. Preemptive scheduling of uniform machines by ordinary network flow techniques. *Management Science* **32**, 341–349.

FERNANDEZ-BACA, D., and C. U. MARTEL. 1989. On the efficiency of maximum flow algorithms on networks with small integer capacities. *Algorithmica* **4**, 173–189.

FILLIBEN, J. J., K. KAFADAR, and D. R. SHIER. 1983. Testing for homogeneity of two-dimensional surfaces. *Mathematical Modelling* **4**, 167–189.

FISHER, M. L. 1981. The Lagrangian relaxation methods for solving integer programming problems. *Management Science* **27**, 1–18.

FISHER, M. L. 1985. An applications oriented guide to Lagrangian relaxation. *Interfaces* **15**, 10–21.

FLORIAN, M. 1986. Nonlinear cost network models in transportation analysis. *Mathematical Programming Study* **26**, 167–196.

FLOYD, R. W. 1962. Algorithm 97: Shortest path. *Communications of ACM* **5**, 345.

FORD, L. R. 1956. Network flow theory. Report P-923, Rand Corp., Santa Monica, CA.

FORD, L. R., and D. R. FULKERSON. 1956a. Maximal flow through a network. *Canadian Journal of Mathematics* **8**, 399–404.

FORD, L. R., and D. R. FULKERSON. 1956b. Solving the transportation problem. *Management Science* **3**, 24–32.

FORD, L. R., and D. R. FULKERSON. 1957. A primal–dual algorithm for the capacitated Hitchcock problem. *Naval Research Logistics Quarterly* **4**, 47–54.

FORD, L. R., and D. R. FULKERSON. 1958a. Constructing maximum dynamic flows from static flows. *Operations Research* **6**, 419–433.

FORD, L. R., and D. R. FULKERSON. 1958b. A suggested computation for maximal multicommodity network flows. *Management Science* **5**, 97–101.

FORD, L. R., and D. R. FULKERSON. 1962. *Flows in Networks*. Princeton University Press, Princeton, NJ.

FORD, L. R., and S. M. JOHNSON. 1959. A tournament problem. *The American Mathematical Monthly* **66**, 387–389.

FRANCIS, R. L., and J. A. WHITE. 1976. *Facility Layout and Location*. Prentice Hall, Englewood Cliffs, NJ.

FRANK, C. R. 1965. A note on the assortment problem. *Management Science* **11**, 724–726.

FRANK, H., and I. T. FRISCH. 1971. *Communication, Transmission, and Transportation Networks*. Addison-Wesley, Reading, MA.

FREDMAN, M. L., and R. E. TARJAN. 1984. Fibonacci heaps and their uses in improved network optimization algorithms. *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, pp. 338–346. Full paper in *Journal of ACM* **34**(1987), 596–615.

FUJII, M., T. KASAMI, and K. NINOMIYA. 1969. Optimal sequencing of two equivalent processors. *SIAM Journal on Applied Mathematics* **17**, 784–789. Erratum, same journal **18**, 141.

FUJISHIGE, S. 1986. A capacity-rounding algorithm for the minimum cost circulation problem: A dual framework of Tardos' algorithm. *Mathematical Programming* **35**, 298–308.

FULKERSON, D. R. 1961a. A network flow computation for project cost curve. *Management Science* **7**, 167–178.

FULKERSON, D. R. 1961b. An out-of-kilter method for minimal cost flow problems. *SIAM Journal on Applied Mathematics* **9**, 18–27.

FULKERSON, D. R. 1963. Flows in networks. In *Recent Advances in Mathematical Programming*, edited by R. L. Graves and P. Wolfe. McGraw-Hill, New York, pp. 319–332.

FULKERSON, D. R. 1965. Upsets in a round robin tournament. *Canadian Journal of Mathematics* **17**, 957–969.

FULKERSON, D. R. 1966. Flow networks and combinatorial operations research. *American Mathematical Monthly* **73**, 115–138.

FULKERSON, D. R., and G. B. DANTZIG. 1955. Computation of maximum flow in networks. *Naval Research Logistics Quarterly* **2**, 277–283.

FULKERSON, D. R., and G. C. HARDING. 1977. Maximizing the minimum source–sink path subject to a budget constraint. *Mathematical Programming* **13**, 116–118.

GABOW, H. N. 1975. An efficient implementation of Edmond's algorithm for maximum matchings on graphs. *Journal of ACM* **23**, 221–234.

GABOW, H. N. 1985. Scaling algorithms for network problems. *Journal of Computer and System Sciences* **31**, 148–168.

GABOW, H. N. 1990. Data structures for weighted matching and nearest common ancestors with linking. *Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms.* SIAM, Philadelphia, pp. 434–443.

GABOW, H. N., Z. GALIL, T. SPENCER, and R. E. TARJAN. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* **6**, 109–122.

GABOW, H. N., and R. E. TARJAN. 1989a. Faster scaling algorithms for network problems. *SIAM Journal on Computing* **18**, 1013–1036.

GABOW, H. N., and R. E. TARJAN. 1989b. Faster scaling algorithms for general graph matching problems. Technical Report CU-CS-432-89, Department of Computer Science, University of Colorado, Boulder, CO.

GALE, D. 1957. A theorem on flows in networks. *Pacific Journal of Mathematics* **7**, 1073–1082.

GALE, D., and L. S. SHAPLEY. 1962. College admissions and the stability of marriage. *American Mathematical Monthly* **69**, 9–14.

GALIL, Z. 1981. On the theoretical efficiency of various network flow algorithms. *Theoretical Computer Science* **14**, 103–111.

GALIL, Z., and É. TARDOS. 1986. An $O(n^2(m + n \log n) \log n)$ min-cost flow algorithm. *Proceedings of the 27th Annual Symposium on the Foundations of Computer Science*, pp. 136–146. Full paper in *Journal of ACM* **35**(1987), 374–386.

GALLO, G., M. D. GRIGORIADIS, and R. E. TARJAN. 1989. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing* **18**, 30–55.

GALLO, G., and S. PALLOTTINO. 1984. Shortest path methods in transportation models. In *Transportation Planning Models*, edited by M. Florian. Elsevier/North-Holland, Amsterdam.

GALLO, G., and S. PALLOTTINO. 1986. Shortest path methods: A unifying approach. *Mathematical Programming Study* **26**, 38–64.

GALLO, G., and S. PALLOTTINO. 1988. Shortest path algorithms. In *Fortran Codes for Network Optimization*, edited by B. Simeone, P. Toth, G. Gallo, F. Maffioli, and S. Pallottino. *Annals of Operations Research* **13**, 3–79.

GAREY, M. S., and D. S. JOHNSON. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York.

GAVISH, B. 1985. Augmented Lagrangian based algorithms for centralized network design. *IEEE Transactions on Communications* **COM-33**, 1247–1257.

GAVISH, B., and P. SCHWEITZER. 1974. An algorithm for combining truck trips. *Transportation Science* **8**, 13–23.

GAVISH, B., and K. N. SRIKANTH. 1979. $O(n^2)$ algorithms for sensitivity analysis of minimal spanning trees and related subgraphs. Working Paper 8003, Graduate School of Management, University of Rochester, Rochester, NY.

GEOFFRION, A. 1974. Lagrangian relaxations for integer programming. *Mathematical Programming Study* **2**, 82–114.

GEOFFRION, A. M., and G. W. GRAVES. 1974. Multicommodity distribution system design by Benders decomposition. *Management Science* **20,** 822–844.

GERSHT, A., and A. SHULMAN. 1987. A new algorithm for the solution of the minimum cost multicommodity flow problem. *Proceedings of the IEEE Conference on Decision and Control* **26,** 748–758.

GILMORE, P. C., and R. E. GOMORY. 1964. Sequencing a one state-variable machine: A solvable case of the travelling salesman problem. *Operations Research* **12,** 655–679.

GLOVER, F., R. GLOVER, and F. K. MARTINSON. 1984. A netform system for resource planning in the U.S. Bureau of Land Management. *Journal of the Operational Research Society* **35,** 605–616.

GLOVER, F., R. GLOVER, and D. J. SHIELDS. 1988. Microcomputer-based model of international mineral market. In *Operational Research '87*, edited by G. K. Rand. Elsevier, Amsterdam.

GLOVER, F., J. HULTZ, D. KLINGMAN, and J. STUTZ. 1978. Generalized networks: A fundamental computer based planning tool. *Management Science* **24,** 1209–1220.

GLOVER, F., D. KARNEY, and D. KLINGMAN. 1974. Implementation and computational comparisons of primal, dual and primal–dual computer codes for minimum cost network flow problem. *Networks* **4,** 191–212.

GLOVER, F., D. KARNEY, D. KLINGMAN, and A. NAPIER. 1974. A computational study on start procedures, basis change criteria, and solution algorithms for transportation problem. *Management Science* **20,** 793–813.

GLOVER, F., and D. KLINGMAN. 1977. Network applications in industry and government. *AIIE Transactions* **9,** 363–376.

GLOVER, F., D. KLINGMAN, J. MOTE, and D. WHITMAN. 1984. A primal simplex variant for the maximum flow problem. *Naval Research Logistics Quarterly* **31,** 41–61.

GLOVER, F., D. KLINGMAN, and N. PHILLIPS. 1985. A new polynomially bounded shortest path algorithm. *Operations Research* **33,** 65–73.

GLOVER, F., D. KLINGMAN, and N. PHILLIPS. 1990. Netform modeling and applications. *Interfaces* **20,** 7–27.

GLOVER, F., D. KLINGMAN, N. PHILLIPS, and R. F. SCHNEIDER. 1985. New polynomial shortest path algorithms and their computational attributes. *Management Science* **31,** 1106–1128.

GLOVER, F., and J. ROGOZINSKI. 1982. Resort development: A network-related model for optimizing sites and visits. *Journal of Leisure Research*, 235–247.

GOETSCHALCKX, M., and H. D. RATLIFF. 1988. Order picking in an aisle. *IIE Transactions* **20,** 53–62.

GOLDBERG, A. V. 1985. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT, Cambridge, MA.

GOLDBERG, A. V., M. D. GRIGORIADIS, and R. E. TARJAN. 1988. Efficiency of the network simplex algorithm for the maximum flow problem. Technical Report, Department of Computer Science, Stanford University, Stanford, CA.

GOLDBERG, A. V., S. A. PLOTKIN, and É. TARDOS. 1991. Combinatorial algorithms for the generalized circulation problem. *Mathematics of Operations Research* **16,** 351–381.

GOLDBERG, A. V., É. TARDOS, and R. E. TARJAN. 1989. Network flow algorithms. Technical Report 860, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY.

GOLDBERG, A. V., and R. E. TARJAN. 1986. A new approach to the maximum flow problem. *Proceedings of the 18th ACM Symposium on the Theory of Computing*, pp. 136–146. Full paper in *Journal of ACM* **35**(1988), 921–940.

GOLDBERG, A. V., and R. E. TARJAN. 1987. Solving minimum cost flow problem by successive approximation. *Proceedings of the 19th ACM Symposium on the Theory of Computing*, pp. 7–18. Full paper in *Mathematics of Operations Research* **15**(1990), 430–466.

GOLDBERG, A. V., and R. E. TARJAN. 1988. Finding minimum-cost circulations by cancelling negative cycles. *Proceedings of the 20th ACM Symposium on the Theory of Computing*, pp. 388–397. Full paper in *Journal of ACM* **36**(1989), 873–886.

GOLDEN, B. L. 1975. A minimum cost multicommodity network flow problem concerning imports and exports. *Networks* **5,** 331–356.

GOLDEN, B. L., A. A. ASSAD, E. A. WASIL, and E. BAKER. 1986. Experiments in optimization. Working Paper Series MS/S 86-004, University of Maryland, College Park, MD.

GOLDEN, B. L., M. LIBERATORE, and C. LIEBERMAN. 1979. Models and solution techniques for cash flow management. *Computers and Operations Research* **6,** 13–20.

GOLDEN, B. L., and T. L. MAGNANTI. 1977. Deterministic network optimization: A bibliography. *Networks* **7,** 149–183.

GOLDFARB, D. 1985. Efficient dual simplex algorithms for the assignment problem. *Mathematical Programming* **33,** 187–203.

GOLDFARB, D., and J. HAO. 1988. Polynomial-time primal simplex algorithms for the minimum cost network flow problem. Technical Report, Department of Industrial Engineering and Operations Research, Columbia University, New York.

GOLDFARB, D., and J. HAO. 1990. A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time. *Mathematical Programming* **47,** 353–365.

GOLDFARB, D., J. HAO, and S. KAI. 1990a. Efficient shortest path simplex algorithms. *Operations Research* **38,** 624–628.

GOLDFARB, D., J. HAO, and S. KAI. 1990b. Anti-stalling pivot rules for the network simplex algorithm. *Networks* **20,** 79–91.

GOLDMAN, A. J., and G. L. NEMHAUSER. 1967. A transport improvement problem transformable to a best-path problem. *Transportation Science* **1,** 295–307.

GOLITSCHEK, M. V., and H. SCHNEIDER. 1984. Applications of shortest path algorithms to matrix scalings. *Numerische Mathematik* **44,** 111–126.

GOMORY, R. E., and T. C. HU. 1961. Multi-terminal network flows. *Journal of SIAM* **9,** 551–570.

GONDRAN, M., and M. MINOUX. 1984. *Graphs and Algorithms*. Wiley-Interscience, New York.

GORHAM, W. 1963. An application of a network flow model to personnel planning. *IEEE Transactions on Engineering Management* **10,** 113–123.

GOWER, J. C., and G. J. S. ROSS. 1969. Minimum spanning trees and single linkage cluster analysis. *Applied Statistics* **18,** 54–64.

GRAHAM, R. L., and P. HELL. 1985. On the history of minimum spanning tree problem. *Annals of the History of Computing* **7,** 43–57.

GRAVES, S. C. 1982. Using Lagrangian techniques to solve hierarchical production planning problems. *Management Science* **28,** 260–275.

GRAVES, G. W., and R. D. MCBRIDE. 1976. The factorization approach to large scale linear programming. *Mathematical Programming* **10,** 91–110.

GREENBERG, H. 1990. Computational testing: Why, how, and how much. *ORSA Journal of Computing* **2,** 94–97.

GRIGORIADIS, M. D. 1986. An efficient implementation of the network simplex method. *Mathematical Programming Study* **26,** 83–111.

GRIGORIADIS, M. D., and Y. HSU. 1979. The Rutgers minimum cost network flow subroutines. *SIGMAP Bulletin of the ACM* **26,** 17–18.

GRÖTSCHEL, M., and O. HOLLAND. 1985. Solving matching problems with linear programming. *Mathematical Programming* **33,** 243–259.

GUIGNARD, M., and S. KIM. 1987a. Lagrangian decomposition: A model yielding stronger Lagrangian bounds. *Mathematical Programming* **39,** 215–228.

GUIGNARD, M., and S. KIM. 1987b. Lagrangian decomposition for integer programming: Theory and applications. Technical Report 93, Department of Statistics, The Wharton School, University of Pennsylvania, Philadelphia, PA.

GUPTA, S. K. 1985. *Linear Programming and Network Models*. Affiliated East–West Press, New Delhi, India.

GUSFIELD, D. 1988. A graph theoretic approach to statistical data security. *SIAM Journal on Computing* **17,** 552–571.

GUSFIELD, D. 1990. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing* **19,** 143–155.

GUSFIELD, D., and R. W. IRVING. 1989. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, Cambridge, MA.

GUSFIELD, D., and C. MARTEL. 1989. A fast algorithm for the generalized parametric minimum cut problem and applications. Technical Report CSE-89-21, Computer Science Division, University of California, Davis, CA.

GUSFIELD, D., C. MARTEL, and D. FERNANDEZ-BACA. 1987. Fast algorithms for bipartite network flow. *SIAM Journal on Computing* **16,** 237–251.

GUTJAHR, A. L., and G. L. NEMHAUSER. 1964. An algorithm for the line balancing problem. *Management Science* **11,** 308–315.

HALL, M. 1956. An algorithm for distinct representatives. *American Mathematical Monthly* **63,** 716–717.

HAMACHER, H. W., and S. TUFEKCI. 1987. On the use of lexicographic min cost flows in evacuation modelling. *Naval Research Logistics Quarterly* **34,** 487–504.

HANDLER, G. Y. 1973. Minimax location of a facility in an undirected graph. *Transportation Science* **7,** 287–293.

HANDLER, G., and I. ZANG. 1980. A dual algorithm for the constrained shortest path problem. *Networks* **10,** 293–309.

HASSIN, R. 1981. Maximum flow in (s, t)-planar networks. *Information Processing Letters* **13,** 107.

HASSIN, R., and D. B. JOHNSON. 1985. An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM Journal on Computing* **14,** 612–624.

HAUSMAN, H. 1978. *Integer Programming and Related Areas: A Classified Bibliography*. Lecture Notes in Economics and Mathematical Systems, Vol. 160. Springer-Verlag, Berlin.

HAX, A. C., and C. CANDEA. 1984. *Production and Inventory Management*. Prentice Hall, Englewood Cliffs, NJ.

HAYMOND, R. E., J. P. JARVIS, and D. R. SHIER. 1980. Computational methods for minimum spanning tree problems. Technical Report 354, Department of Mathematical Sciences, Clemson University, Clemson, SC.

HAYMOND, R. E., J. R. THORNTON, and D. D. WARNER. 1988. A shortest path algorithm in robotics and its implementation on the FPS T-20 hypercube. *Annals of Operations Research* **14,** 305–320.

HELD, M., and R. KARP. 1970. The traveling salesman problem and minimum spanning trees. *Operations Research* **18,** 1138–1162.

HELD, M., and R. KARP. 1971. The traveling salesman problem and minimum spanning trees, Part II. *Mathematical Programming* **6,** 62–88.

HELGASON, R. V., J. L. KENNINGTON, and B. D. STEWART. 1988. Dijkstra's two-tree shortest path algorithm. Technical Report, Department of Operations Research and Engineering Management, Southern Methodist University, Dallas, TX.

HITCHCOCK, F. L. 1941. The distribution of a product from several sources to numerous facilities. *Journal of Mathematical Physics* **20,** 224–230.

HOCHBAUM, D. S., and J. G. SHANTHIKUMAR. 1990. Convex separable optimization is not much harder than linear optimization. *Journal of ACM* **37,** 843–862.

HOFFMAN. A. J. 1960. Some recent applications of the theory of linear inequalities to extremal combinatorial analysis. In *Combinatorial Analysis*, edited by R. Bellman and M. Hall. American Mathematical Society, Providence, RI, pp. 113–128.

HOFFMAN, K. L., and R. H. F. JACKSON. 1982. In pursuit of a methodology for testing mathematical programming software. In *Evaluating Mathematical Programming Techniques*, Lecture Notes in Economics and Mathematical Systems, Vol. 199, edited by J. M. Mulvey et al., Springer-Verlag, New York.

HOFFMAN, A. J., and J. B. KRUSKAL. 1956. Integral boundary points of convex polyhedra. In *Linear Inequalities and Related Systems*, edited by H. W. Kuhn and A. W. Tucker. Princeton University Press, Princeton, NJ, pp. 233–246.

HOFFMAN, A. J., and H. M. MARKOWITZ. 1963. A note on shortest path, assignment and transportation problems. *Naval Research Logistics Quarterly* **10,** 375–379.

HOFFMAN, A. J., and S. T. MCCORMICK. 1984. A fast algorithm that makes matrices optimally sparse. In *Progress in Combinatorial Optimization*. Academic Press Canada, Don Mills, Ontario, Canada.

HOPCROFT, J. E., and R. M. KARP. 1973. A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing* **2,** 225–231.

HOPCROFT, J. E., and R. E. TARJAN. 1974. Efficient planarity testing. *Journal of ACM* **21,** 549–568.

HORN, W. A. 1971. Determining optimal container inventory and routing. *Transportation Science* **5,** 225–231.

HORN, W. A. 1973. Minimizing average flow time with parallel machines. *Operations Research* **21,** 846–847.

HU, T. C. 1961. The maximum capacity route problem. *Operations Research* **9,** 898–900.

HU, T. C. 1963. Multi-commodity network flows. *Operations Research* **11,** 344–360.

HU, T. C. 1966. Minimum cost flows in convex cost networks. *Naval Research Logistics Quarterly* **13,** 1–9.

HU, T. C. 1967. Laplace's equation and network flows. *Operations Research* **15,** 348–354.

HU, T. C. 1969. *Integer Programming and Network Flows*. Addison-Wesley, Reading, MA.

HU, T. C. 1974. Optimum communication spanning trees. *SIAM Journal on Computing* **3,** 188–195.

HUNG, M. S. 1983. A polynomial simplex method for the assignment problem. *Operations Research* **31,** 595–600.

HUNG, M. S., and J. J. DIVOKY. 1988. A computational study of efficient shortest path algorithms. *Computers and Operations Research* **15,** 567–576.

IMAI, H. 1983. On the practical efficiency of various maximum flow algorithms. *Journal of the Operations Research Society of Japan* **26,** 61–82.

IMAI, H., and M. IRI. 1986. Computational-geometric methods for polygonal approximations of a curve. *Computer Vision, Graphics and Image Processing* **36,** 31–41.

IRI, M. 1960. A new method of solving transportation-network problems. *Journal of the Operations Research Society of Japan* **3,** 27–87.

IRI, M. 1969. *Network Flow, Transportation and Scheduling.* Academic Press, New York.

ITAI, A., and Y. SHILOACH. 1979. Maximum flow in planar networks. *SIAM Journal on Computing* **8,** 135–150.

JACKSON, R. H. B., P. T. BOGGS, S. G. NASH, and S. POWELL. 1989. Report of the ad hoc committee to revise the guidelines for reporting computational experiments in mathematical programming. *COAL Newsletter* **18,** 3–14.

JACKSON, R. H. B., and J. M. MULVEY. 1978. A critical review of comparisons of mathematical programming algorithms and software (1953–1977). *Journal of Research of the National Bureau of Standards* **83,** 563–584.

JACOBS, W. W. 1954. The caterer problem. *Naval Research Logistics Quarterly* **1,** 154–165.

JARNÍCK, V. 1930. O jistém problému minimálním. *Acta Societatis Scientiarum Natur. Moravicae* **6,** 57–63.

JARVIS, J. P., and D. E. WHITE. 1983. Computational experience with minimum spanning tree algorithms. *Operations Research Letters* **2,** 36–41.

JENSEN, P. A., and W. BARNES. 1980. *Network Flow Programming.* Wiley, New York.

JENSEN, P., and G. BHAUMIK. 1977. A flow augmentation approach to the network with gains minimum cost flow problem. *Management Science* **23,** 631–643.

JEWELL, W. S. 1957. Warehousing and distribution of a seasonal product. *Naval Research Logistics Quarterly* **4,** 29–34.

JEWELL, W. S. 1958. Optimal flow through networks. Interim Technical Report 8, Operations Research Center, MIT, Cambridge, MA.

JEWELL, W. S. 1962. Optimal flow through networks with gains. *Operations Research* **10,** 476–499.

JOHNSON, E. L. 1966. Networks and basic solutions. *Operations Research* **14,** 619–624.

JOHNSON, T. B. 1968. Optimum pit mine production scheduling. Technical Report, University of California, Berkeley, CA.

JOHNSON, D. B. 1982. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Mathematical Systems Theory* **15,** 295–309.

JOHNSON, D. S. 1990. Local optimization and the traveling salesman problem. *Proceedings of the 17th Colloquium on Automata, Languages, and Programming.* Springer-Verlag, New York, pp. 446–461.

JOHNSON, D. B., and S. VENKATESAN. 1982. Using divide and conquer to find flows in directed planar networks in $O(n^{3/2} \log n)$ time. *Proceedings of the 20th Annual Allerton Conference on Communication, Control, and Computing,* University of Illinois, Urbana-Champaign, IL, pp. 898–905.

KAMEDA, T., and I. MUNRO. 1974. A $O(|V| \cdot |E|)$ algorithm for maximum matching of graphs. *Computing* **12,** 91–98.

KANG, A. N. C., R. C. T. LEE, C. L. CHANG, and S. K. CHANG. 1977. Storage reduction through minimal spanning trees and spanning forests. *IEEE Transactions on Computers* **C-26,** 425–434.

KANTOROVICH, L. V. 1939. Mathematical methods in the organization and planning of production. Publication House of the Leningrad University. Translated in *Management Science* **6**(1960), 366–422.

KAPLAN, S. 1973. Readiness and the optimal redeployment of resources. *Naval Research Logistics Quarterly* **20,** 625–638.

KARP, R. M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations,* edited by R. E. Miller and J. W. Thacher. Plenum Press, New York, pp. 83–103.

KARP, R. M. 1978. A characterization of the minimum cycle mean in a diagraph. *Discrete Mathematics* **23,** 309–311.

KARP, R. M., and J. B. ORLIN. 1981. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete Applied Mathematics* **3**, 37–45.

KARZANOV, A. V. 1974. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady* **15**, 434–437.

KASTNING, C. 1976. *Integer Programming and Related Areas: A Classified Bibliography*. Lecture Notes in Economics and Mathematical Systems, Vol. 128, Springer-Verlag, Berlin.

KELLY, J. R. 1961. Critical path planning and scheduling: Mathematical basis. *Operations Research* **9**, 296–320.

KELLY, J. P., B. L. GOLDEN, and A. A. ASSAD. 1992. Cell suppression: Disclosure protection for sensitive tabular data. *Networks* **22**, 397–412.

KENNINGTON, J. L. 1978. A survey of linear cost multicommodity network flows. *Operations Research* **26**, 209–236.

KENNINGTON, J. L., and R. V. HELGASON. 1980. *Algorithms for Network Programming*. Wiley-Interscience, New York.

KENNINGTON, J. L., and M. SHALABY. 1977. An effective subgradient procedure for minimal cost multicommodity flow problems. *Management Science* **23**, 994–1004.

KENNINGTON, J. L., and Z. WANG. 1990. The shortest augmenting path algorithm for the transportation problem. Technical Report 90-CSE-10, Southern Methodist University, Dallas, TX.

KHAN, M. R. 1979. A capacitated network formulation for manpower scheduling. *Industrial Management* **21**, 24–28.

KHAN, M. R., and D. A. LEWIS. 1987. A network model for nursing staff scheduling. *Zeitschrift für Operations Research* **31**, B161–B171.

KLEIN, M. 1967. A primal method for minimal cost flows with application to the assignment and transportation problems. *Management Science* **14**, 205–220.

KLINCEWICZ, J. G. 1983. A Newton method for convex separable network flow problems. *Networks* **13**, 427–442.

KLINGMAN, D., A. NAPIER, and J. STUTZ. 1974. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science* **20**, 814–821.

KNUTH, D. E. 1973a. *The Art of Computer Programming*. Vol. 1: *Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, MA.

KNUTH, D. E. 1973b. *The Art of Computer Programming*. Vol. III: *Sorting and Searching*. Addison-Wesley, Reading, MA.

KOLITZ, S. 1991. Personal communication.

KOOPMANS, T. C. 1947. Optimum utilization of the transportation system. *Proceedings of the International Statistical Conference*, Washington, DC. Also in *Econometrica* **17**(1949).

KORTE, B. 1988. Applications of combinatorial optimization. Technical Report 88541-OR, Institute für Okonometrie und Operations Research, Bonn, Germany.

KOURTZ, P. 1984. A network approach to least cost daily transfers of forest fire control resources. *INFOR* **22**, 283–290.

KRUSKAL, J. B. 1956. On the shortest spanning tree of graph and the traveling salesman problem. *Proceedings of the American Mathematical Society* **7**, 48–50.

KUHN, H. W. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* **2**, 83–97.

LAPORTE, G., and Y. NOBERT. 1987. Exact algorithms for the vehicle routing problem. In *Surveys in Combinatorial Optimization*, edited by S. Martello, G. Laporte, M. Minoux, and C. Ribeiro. North-Holland, Amsterdam.

LARSON, R. C., and A. R. ODONI. 1981. *Urban Operations Research*. Prentice Hall, Englewood Cliffs, NJ.

LAWANIA, A. K. 1990. Personal communication.

LAWLER, E. L. 1964. On scheduling problems with deferral costs. *Management Science* **11**, 280–287.

LAWLER, E. L. 1966. Optimal cycles in doubly weighted linear graphs. In *Theory of Graphs: International Symposium*, Dunod, Paris, and Gordon and Breach, New York, pp. 209–213.

LAWLER, E. L. 1976. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York.

LAWLER, E. L., J. K. LENSTRA, A. H. G. RINNOOY KAN, and D. B. SHMOYS (eds.). 1985. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, New York.

LEUNG, J., T. L. MAGNANTI, and V. SINGHAL. 1990. Routing in point to point delivery systems. *Transportation Science* **24,** 245–260.

LEVIN, L. A. 1973. Universal sorting problems. *Problemy Peredachi Informatsii* **9,** 265–266. (In Russian.)

LEVNER, E. V., and A. S. NEMIROVSKY. 1991. A network flow algorithm for just-in-time project scheduling. Memorandum COSOR 91-21, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

LIN, T. F. 1986. A system of linear equations related to the transportation problem with application to probability theory. *Discrete Applied Mathematics* **14,** 47–56.

LOBERMAN, H., and A. WEINBERGER. 1957. Formal procedures for connecting terminals with a minimum total wire length. *Journal of ACM* **4,** 428–437.

LOVÁSZ, L., and M. D. PLUMMER. 1986. *Matching Theory.* North-Holland, Amsterdam.

LOVE, R. R., and R. R. VEMUGANTI. 1978. The single-plant mold allocation problem with capacity and changeover restriction. *Operations Research* **26,** 159–165.

LOWE, T. J., R. L. FRANCIS, and E. W. REINHARDT. 1979. A greedy network flow algorithm for a warehouse leasing problem. *AIIE Transactions* **11,** 170–182.

LUSS, H. 1979. A capacity expansion model for two facilities. *Naval Research Logistics Quarterly* **26,** 291–303.

MACHOL, R. E. 1961. An application of the assignment problem. *Operations Research* **9,** 585–586.

MACHOL, R. E. 1970. An application of the assignment problem. *Operations Research* **18,** 745–746.

MAGNANTI, T. L. 1981. Combinatorial optimization and vehicle fleet planning: Perspectives and prospects. *Networks* **11,** 179–214.

MAGNANTI, T. L. 1984. Models and algorithms for predicting urban traffic equilibria. In *Transportation Planning Models*, edited by M. Florian. North-Holland, Amsterdam, pp. 153–186.

MAGNANTI, T. L., P. MIRCHANDANI, and R. VACHANI. 1991. Modeling and solving the capacitated network loading problem. Working Paper, Operations Research Center, MIT, Cambridge, MA.

MAGNANTI, T. L., J. SHAPIRO, and M. WAGNER. 1976. Generalized linear programming solves the dual. *Management Science* **22,** 1195–1203.

MAGNANTI, T. L., L. A. WOLSEY, and R. T. WONG. 1992. Optimal Trees. To appear in *Handbooks in Operations Research and Management Science.* Vol. 6: *Networks*, edited by M. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser. North-Holland, Amsterdam.

MAGNANTI, T. L., L. A. WOLSEY, and R. T. WONG. 1992. Network design. To appear in *Handbooks in Operations Research and Management Science*, Vol. 6: *Networks*, edited by M. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser. North-Holland, Amsterdam.

MAGNANTI, T. L., and R. T. WONG. 1984. Network design and transportation planning: Models and algorithms. *Transportation Science* **18,** 1–55.

MALIK, K., A. K. MITTAL, and S. K. GUPTA. 1989. The k most vital arcs in the shortest path problem. *Operations Research Letters* **8,** 223–227. Erratum: Same journal **9**(1990) 283.

MAMER, J. W., and S. A. SMITH. 1982. Optimizing field repair kits based on job completion rate. *Management Science* **28,** 1328–1334.

MANNE, A. S. 1958. A target-assignment problem. *Operations Research* **6,** 346–351.

MANSOUR, Y., and B. SCHIEBER. 1988. Finding the edge connectivity of directed graphs. Research Report RC 13556, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

MARTEL, C. 1982. Preemptive scheduling with release times, deadlines, and due times. *Journal of ACM* **29,** 812–829.

MARTELLO, S., W. R. PULLEYBLANK, P. TOTH, and D. DE WERRA. 1984. Balanced optimization problems. *Operations Research Letters* **3,** 275–278.

MASON, A. J., and A. B. PHILPOTT. 1988. Pairing stereo speakers using matching algorithms. *Asia-Pacific Journal of Operational Research* **5,** 101–116.

MATSUMOTO, K., T. NISHIZEKI, and N. SAITO. 1985. An efficient algorithm for finding multicommodity flows in planar networks. *SIAM Journal on Computing* **14,** 289–302.

MATULA, D. W. 1987. Determining edge connectivity in $O(nm)$. *Proceedings of the 28th Symposium on Foundations of Computer Science*, pp. 249–251.

MAXWELL, W. L., and R. C. WILSON. 1981. Dynamic network flow modelling of fixed path material handling systems. *AIIE Transactions* **13,** 12–21.

McGeoch, C. C. 1986. *Experimental Analysis of Algorithms*. Unpublished Ph.D. dissertation, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

McGeoch, C. C. 1992. Analysis of algorithms by simulation: Variance reduction techniques and simulation speedups. *Computing Surveys* **24,** June issue.

McGinnis, L. F., and H. L. W. Nuttle. 1978. The project coordinators problem. *OMEGA* **6,** 325–330.

Meggido, N. 1979. Combinatorial optimization with rational objective functions. *Mathematics of Operations Research* **4,** 414–424.

Meggido, N., and A. Tamir. 1978. An $O(n \log n)$ algorithm for a class of matching problems. *SIAM Journal on Computing* **7,** 154–157.

Mehlhorn, K. 1984. *Data Structures and Algorithms*, Vol. I: *Searching and Sorting*. Springer-Verlag, New York.

Micali, S., and V. V. Vazirani. 1980. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. *Proceedings of the 21st Annual Symposium on the Foundations of Computer Science*, pp. 17–27.

Minieka, E. 1978. *Optimization Algorithms for Networks and Graphs*. Marcel Dekker, New York.

Minoux, M. 1984. A polynomial algorithm for minimum quadratic cost flow problems. *European Journal of Operational Research* **18,** 377–387.

Minoux, M. 1986. Solving integer minimum cost flows with separable convex cost objective polynomially. *Mathematical Programming Study* **26,** 237–239.

Minoux, M. 1989. Network synthesis and optimal network design problems: Models, solution methods, and applications. *Networks* **19,** 313–360.

Minty, G. J. 1960. Monotone networks. *Proceedings of the Royal Society of London* **257A,** 194–212.

Monma, C. L., and M. Segal. 1982. A primal algorithm for finding minimum-cost flows in capacitated networks with applications. *Bell System Technical Journal* **61,** 449–468.

Moore, E. F. 1957. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching, Part II;* The Annals of the Computation Laboratory of Harvard University **30,** Harvard University Press, pp. 285–292.

Mulvey, J. 1978. Pivot strategies for primal–simplex network codes. *Journal of ACM* **25,** 266–270.

Mulvey, J. M. 1979. Strategies in modeling: A personal scheduling example. *Interfaces* **9,** 66–76.

Nemhauser, G. L., and L. A. Wolsey. 1988. *Integer and Combinatorial Optimization*. Wiley, New York.

Orlin, D. 1987. Optimal weapons allocation against layered defenses. *Naval Research Logistics Quarterly* **34,** 605–617.

Orlin, J. B. 1984. Genuinely polynomial simplex and non-simplex algorithms for the minimum cost flow problem. Technical Report 1615-84, Sloan School of Management, MIT, Cambridge, MA.

Orlin, J. B. 1985. On the simplex algorithm for networks and generalized networks. *Mathematical Programming Study* **24,** 166–178.

Orlin, J. B. 1988. A faster strongly polynomial minimum cost flow algorithm. *Proceedings of the 20th ACM Symposium on the Theory of Computing*, pp. 377–387. Full paper to appear in *Operations Research*.

Orlin, J. B., and R. K. Ahuja. 1992. New scaling algorithms for the assignment and minimum cycle mean problems. *Mathematical Programming* **54,** 41–56.

Orlin, J. B., and U. G. Rothblum. 1985. Computing optimal scalings by parametric network algorithms. *Mathematical Programming* **32,** 1–10.

Osteen, R. E., and P. P. Lin. 1974. Picture skeletons based on eccentricities of points of minimum spanning trees. *SIAM Journal on Computing* **3,** 23–40.

Pallottino, S. 1991. Personal communications.

Papadimitriou, C. H., and K. Steiglitz. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ.

Pape, U. 1974. Implementation and efficiency of Moore-algorithms for the shortest route problem. *Mathematical Programming* **7,** 212–222.

Pape, U. 1980. Algorithm 562: Shortest path lengths. *ACM Transactions on Mathematical Software* **6,** 450–455.

Phillips, D. T., and A. Garcia-Diaz. 1981. *Fundamentals of Network Analysis*. Prentice Hall, Englewood Cliffs, NJ.

PICARD, J. C., and M. QUEYRANNE. 1982. Selected applications of minimum cuts in networks. *INFOR* **20,** 394–422.

PICARD, J. C., and H. D. RATLIFF. 1973. Minimal cost cut equivalent networks. *Management Science* **19,** 1087–1092.

PICARD, J. C., and H. D. RATLIFF. 1978. A cut approach to the rectilinear distance facility location problem. *Operations Research* **26,** 422–433.

PINAR, M. C., and S. A. ZENIOS. 1990. Parallel decomposition of multicommodity network flows using smooth penalty functions. Technical Report 90-12-06, Department of Decision Sciences, Wharton School, University of Pennsylvania, Philadelphia, PA.

PINTO, Y., and R. SHAMIR. 1990. Efficient algorithms for minimum cost flow problems with convex costs. Technical Report, Department of Computer Science, Tel Aviv University, Tel Aviv, Israel.

PLOTKIN, S., and É. TARDOS. 1990. Improved dual network simplex. *Proceedings of the First ACM–SIAM Symposium on Discrete Algorithms*, pp. 367–376.

POTTS, R. B., and R. M. OLIVER. 1972. *Flows in Transportation Networks*. Academic Press, New York.

PRAGER, W. 1957. On warehousing problems. *Operations Research* **5,** 504–512.

PRIM, R. C. 1957. Shortest connection networks and some generalizations. *Bell System Technical Journal* **36,** 1389–1401.

RATLIFF, H. D. 1978. Network models for production scheduling problems with convex cost and batch processing. *AIIE Transactions* **10,** 104–108.

RAVINDRAN, A. 1971. On compact book storage in libraries. *Opsearch* **8,** 245–252.

RECSKI, A. 1988. *Matroid Theory and Its Applications*. Springer-Verlag, New York.

RHYS, J. M. W. 1970. A selection problem of shared fixed costs and network flows. *Management Science* **17,** 200–207.

RÖCK, H. 1980. Scaling techniques for minimal cost network flows. In *Discrete Structures and Algorithms*. Edited by V. Page. Carl Hanser, Munich, pp. 181–191.

ROCKAFELLAR, R. T. 1970. *Convex Analysis*. Princeton University Press, Princeton, NJ.

ROCKAFELLAR, R. T. 1984. *Network Flows and Monotropic Optimization*. John Wiley & Sons, New York.

ROOHY-LALEH, E. 1980. Improvements to the Theoretical Efficiency of the Network Simplex Method. Unpublished Ph.D. dissertation, Carleton University, Ottawa, Canada.

ROSENTHAL, R. E. 1981. A nonlinear network flow algorithm for maximization of benefits in a hydroelectric power system. *Operations Research* **29,** 763–786.

ROSS, G. T., and R. M. SOLAND. 1975. A branch and bound algorithm for the generalized assignment problem. *Mathematical Programming* **8,** 91–103.

ROTH, A. E., U. G. ROTHBLUM, and J. H. VANDE VATE. 1990. Stable matchings, optimal assignments and linear programming. Rutcor Research Report 23-90, The State University of New Jersey, Rutgers, NJ.

ROTHFARB, B., N. P. SHEIN, and I. T. FRISCH. 1968. Common terminal multicommodity flow. *Operations Research* **16,** 202–205.

SAKAROVITCH, M. 1973. Two commodity network flows and linear programming. *Mathematical Programming* **4,** 1–20.

SAPOUNTZIS, C. 1984. Allocating blood to hospitals from a central blood bank. *European Journal of Operational Research* **16,** 157–162.

SCHMIDT, S. R., P. A. JENSEN, and J. W. BARNES. 1982. An advanced dual incremental network algorithm. *Networks* **12,** 475–492.

SCHNEIDER, M. H., and S. A. ZENIOS. 1990. A comparative study of algorithms for matrix balancing. *Operations Research* **38,** 439–455.

SCHNEUR, R. 1991. Scaling algorithms for multicommodity flow problems and network flow problems with side constraints. Ph.D. dissertation, Department of Civil Engineering, MIT, Cambridge, MA.

SCHNORR, C. P. 1979. Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM Journal on Computing* **8,** 265–274.

SCHRIJVER, A. 1986. *Theory of Linear and Integer Programming*. Wiley, New York.

SCHWARTZ, B. L. 1966. Possible winners in partially completed tournaments. *SIAM Review* **8,** 302–308.

SCHWARTZ, M., and T. E. STERN. 1980. Routing techniques used in computer communication networks. *IEEE Transactions on Communications* **COM-28,** 539–552.

SEGAL, M. 1974. The operator-scheduling problem: A network flow approach. *Operations Research* **22,** 808–823.

SERVI, L. D. 1989. A network flow approach to a satellite scheduling problem. Research Report, GTE Laboratories, Waltham, MA.

SHAPIRO, J. F. 1979. *Mathematical Programming: Structures and Algorithms*. Wiley, New York.

SHAPIRO, J. F. 1992. Mathematical programming models and methods for production planning and scheduling. To appear in *Handbooks in Operations Research and Management Science*, Vol. 4: *Logistics of Production and Inventory*, edited by S. C. Graves, A. H. G. Rinnooy Kan, and P. Zipkin. North-Holland, Amsterdam.

SHEPARDSON, F., and R. E. MARSTEN. 1980. A Lagrangian relaxation algorithm for the two-duty scheduling problem. *Management Science* **26,** 274–281.

SHIER, D. R. 1982. Testing for homogeneity using minimum spanning trees. *The UMAP Journal* **3,** 273–283.

SHILOACH, Y., and U. VISHKIN. 1982. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms* **3,** 128–146.

SLEATOR, D. D., and R. E. TARJAN. 1983. A data structure for dynamic trees. *Journal of Computer and System Sciences* **24,** 362–391.

SLUMP, C. H., and J. J. GERBRANDS. 1982. A network flow approach to reconstruction of the left ventricle from two projections. *Computer Graphics and Image Processing* **18,** 18–36.

SRINIVASAN, V. 1974. A transshipment model for cash management decisions. *Management Science* **20,** 1350–1363.

SRINIVASAN, V. 1979. Network models for estimating brand-specific effects in multiattribute marketing models. *Management Science* **25,** 11–21.

SRINIVASAN, V., and G. L. THOMPSON. 1972. An operator theory of parametric programming for the transportation problem. *Naval Research Logistics Quarterly* **19,** 205–252.

SRINIVASAN, V., and G. L. THOMPSON. 1973. Benefit–cost analysis of coding techniques for primal transportation algorithm. *Journal of ACM* **20,** 194–213.

STILLINGER, F. H. 1967. Physical clusters, surface tension, and critical phenomenon. *Journal of Chemical Physics* **47,** 2513–2533.

STOER, J., and C. WITZGALL. 1970. *Convexity and Optimization in Finite Dimensions*. Springer-Verlag, New York.

STONE, H. S. 1977. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering* **3,** 85–93.

SWOVELAND, C. 1971. Decomposition algorithms for the multi-commodity distribution problem. Working Paper 184, Western Management Science Institute, University of California, Los Angeles, CA.

SYSLO, M. M., N. DEO, and J. S. KOWALIK. 1983. *Discrete Optimization Algorithms*. Prentice Hall, Englewood Cliffs, NJ.

SZADKOWSKI, S. 1970. An approach to machining process optimization. *International Journal of Production Research* **9,** 371–376.

TALLURI, K. T. 1991. Issues in the design of survivable networks. Ph.D. dissertation, Operations Research Center, MIT, Cambridge, MA.

TARDOS, É. 1985. A strongly polynomial minimum cost circulation algorithm. *Combinatorica* **5,** 247–255.

TARDOS, É. 1986. A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research* **34,** 250–256.

TARJAN, R. E. 1982. Sensitivity analysis of minimum spanning trees and shortest path trees. *Information Processing Letters* **14,** 30–33.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA.

TARJAN, R. E. 1984. A simple version of Karzanov's blocking flow algorithm. *Operations Research Letters* **2,** 265–268.

TARJAN, R. E. 1991. Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem. *Mathematics of Operations Research* **16,** 272–291.

TOMIZAVA, N. 1972. On some techniques useful for solution of transportation network problems. *Networks* **1,** 173–194.

TOMLIN, J. A. 1966. A linear programming model for the assignment of traffic. *Proceedings of the 3rd Conference of the Australian Road Research Board* **3,** 263–271.

TRUEMPER, K. 1977. On max flow with gains and pure min-cost flows. *SIAM Journal on Applied Mathematics* **32,** 450–456.

TSO, M. 1986. Network flow models in image processing. *Journal of the Operational Research Society* **37,** 31–34.

TSO, M., P. KLEINSCHMIDT, I. MITTERREITER, and J. GRAHAM. 1991. An efficient transportation algorithm for automatic chromosome karotyping. *Pattern Recognition Letters* **12,** 117–126.

TUTTE, W. T. 1971. *Introduction to the Theory of Matroids*. American Elsevier, New York.

VAIDYA, P. M. 1989. Speeding up linear programming using fast matrix multiplication. *Proceedings of the 30th Annual Symposium on the Foundations of Computer Science*, pp. 332–337.

VAN SLYKE, R., and H. FRANK. 1972. Network reliability analysis: Part I. *Networks* **1,** 279–290.

VAZIRANI, V. V. 1989. A theory of alternating paths and blossoms for proving correctness of the $O(n^{1/2}m)$ general graph matching algorithm. Technical Report 89-1035, Department of Computer Science, Cornell University, Ithaca, NY.

VEINOTT, A. F., and G. B. DANTZIG. 1968. Integer extreme points. *SIAM Review* **10,** 371–372.

VEINOTT, A. F., and H. M. WAGNER. 1962. Optimal capacity scheduling: Parts I and II. *Operations Research* **10,** 518–547.

VOLGENANT, A. 1989. A Lagrangian approach to the degree-constrained minimum spanning tree problem. *European Journal of Operational Research* **39,** 325–331.

VON RANDOW, R. 1982. *Integer Programming and Related Areas: A Classified Bibliography 1978–1981*. Lecture Notes in Economics and Mathematical Systems, Vol. 197. Springer-Verlag, Berlin.

VON RANDOW, R. 1985. *Integer Programming and Related Areas: A Classified Bibliography 1981–1984*. Lecture Notes in Economics and Mathematical Systems, Vol. 243. Springer-Verlag, Berlin.

WAGNER, R. A. 1976. A shortest path algorithm for edge-sparse graphs. *Journal of ACM* **23,** 50–57.

WAGNER, D. K. 1990. Disjoint (s, t)-cuts in a network. *Networks* **20,** 361–371.

WALLACHER, C., and U. T. ZIMMERMANN. 1991. A combinatorial interior point method for network flow problems. Presented at the *14th International Symposium on Mathematical Programming*, Amsterdam, The Netherlands.

WARSHALL, S. 1962. A theorem on boolean matrices. *Journal of ACM* **9,** 11–12.

WATERMAN, M. S. 1988. *Mathematical Methods for DNA Sequences*. CRC Press, Boca Raton, FL.

WEINTRAUB, A. 1974. A primal algorithm to solve network flow problems with convex costs. *Management Science* **21,** 87–97.

WELSH, D. J. A. 1976. *Matroid Theory*. Academic Press, New York.

WHITE, L. S. 1969. Shortest route models for the allocation of inspection effort on a production line. *Management Science* **15,** 249–259.

WHITE, W. W. 1972. Dynamic transshipment networks: An algorithm and its application to the distribution of empty containers. *Networks* **2,** 211–230.

WHITING, P. D., and J. A. HILLIER. 1960. A method for finding the shortest route through a road network. *Operations Research Quarterly* **11,** 37–40.

WHITNEY, H. 1935. On the abstract properties of linear dependence. *American Journal of Mathematics* **57,** 509–533.

WINSTON, W. L. 1991. *Operations Research: Applications and Algorithms*. PWS-Kent, Boston, MA.

WITZGALL, C., and C. T. ZAHN. 1965. Modification of Edmonds maximum matching algorithm. *Journal of Research of the National Bureau of Standards* **69B,** 91–98.

WONG, R. T. 1980. Integer programming formulations of the traveling salesman problem. *Proceedings of the 1980 IEEE International Conference on Circuits and Computers*, pp. 149–152.

WRIGHT, J. W. 1975. Reallocation of housing by use of network analysis. *Operational Research Quarterly* **26,** 253–258.

YAO, A. 1975. An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters* **4,** 21–23.

YOUNG, N. E., R. E. TARJAN, and J. B. ORLIN. 1990. Faster parametric shortest path and minimum balance algorithms. Working Paper 3112-90-MS, Sloan School of Management, MIT, Cambridge, MA.

ZADEH, N. 1973a. A bad network problem for the simplex method and other minimum cost flow algorithms. *Mathematical Programming* **5,** 255–266.

ZADEH, N. 1973b. More pathological examples for network flow problems. *Mathematical Programming* **5,** 217–224.

ZADEH, N. 1979. Near equivalence of network flow algorithms. Technical Report 26, Department of Operations Research, Stanford University, Stanford, CA.

ZAHN, C. T. 1971. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computing* **C20**, 68–86.

ZAKI, H. 1990. A comparison of two algorithms for the assignment problem. Technical Report ORL 90-002, Department of Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign, Urbana, IL.

ZANGWILL, W. I. 1969. A backlogging model and a multi-echelon model of a dynamic economic lot size production system: A network approach. *Management Science* **15**, 506–527.

ZAWACK, D. J., and G. L. THOMPSON. 1987. A dynamic space–time network flow model for city traffic congestion. *Transportation Science* **21**, 153–162.

ZENIOS, S. A., and J. M. MULVEY. 1986. Relaxation techniques for strictly convex network problems. *Annals of Operations Research* **5**, 517–538.

INDEX