NETWORK FLOWS

THEORY, ALGORITHMS, AND APPLICATIONS (II)

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.

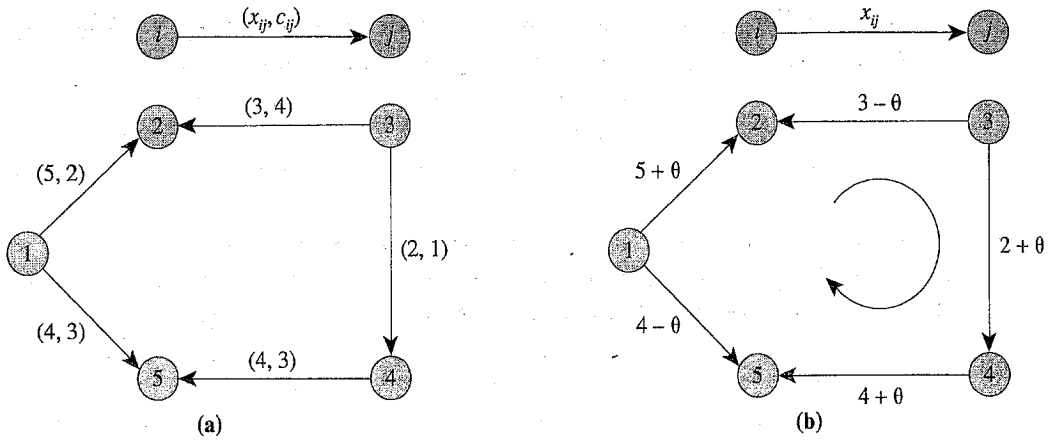


Figure 11.1 Improving flow around a cycle: (a) feasible solution; (b) solution after augmenting θ amount of flow along a cycle.

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.

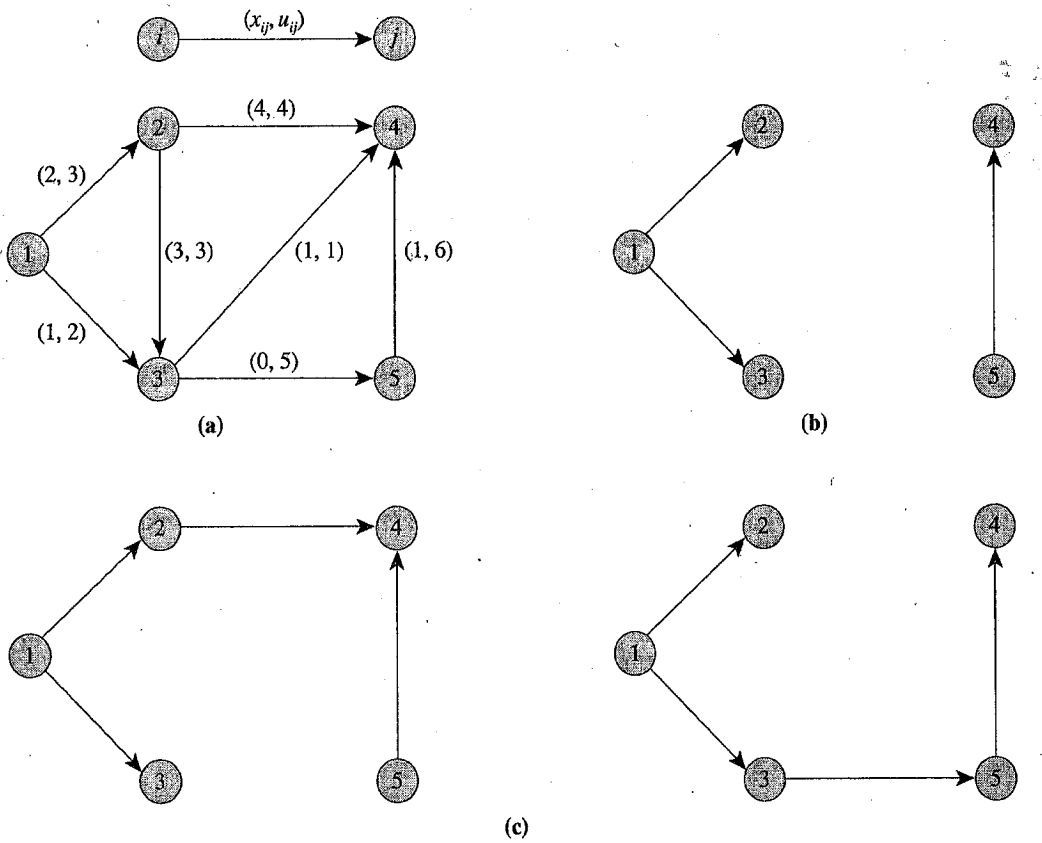


Figure 11.2 Converting a cycle free solution into a spanning tree solution: (a) example network; (b) set of free arcs; (c) 2 spanning tree solutions.

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 $\sum_{(i,j)\in A} c^{\pi}_{ij}x_{ij}$. The conditions stated in (11.1) imply that for the given node potential π, minimizing $\sum_{(i,j)\in A} c^{\pi}_{ij}x_{ij}$ is equivalent to minimizing the following expression:

$$\text{Minimize} \quad \sum_{(i,j)\in \mathbf{L}} c^{\pi}_{ij}x_{ij} - \sum_{(i,j)\in \mathbf{U}} |\, c^{\pi}_{ij} \,|\, x_{ij}. \tag{11.2}$$

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 $c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j)$ 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.

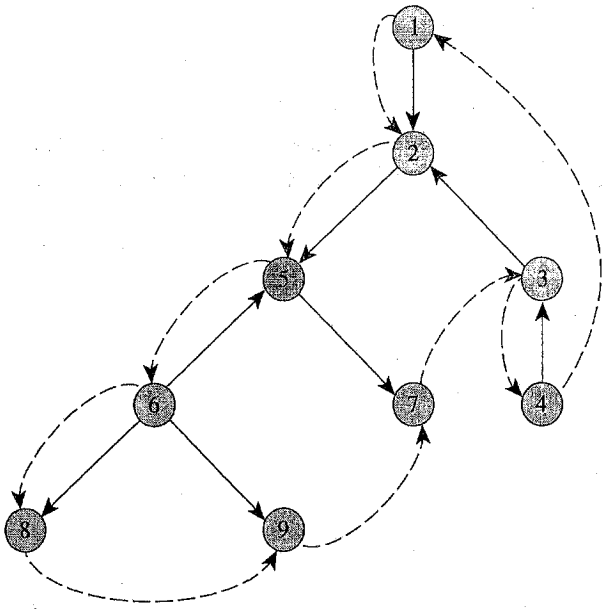
The network simplex algorithm maintains a feasible spanning tree structure and moves from one spanning tree structure to another until it finds an optimal structure. At each iteration, the algorithm adds one arc to the spanning tree in place of one of its current arcs. The entering arc is a nontree arc 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) − c_ij;
        if (j, i) ∈ A then π(j) := π(i) + c_ji;
        j := thread(j);
    end;
end;
```

Figure 11.4 Procedure *compute-potentials*.

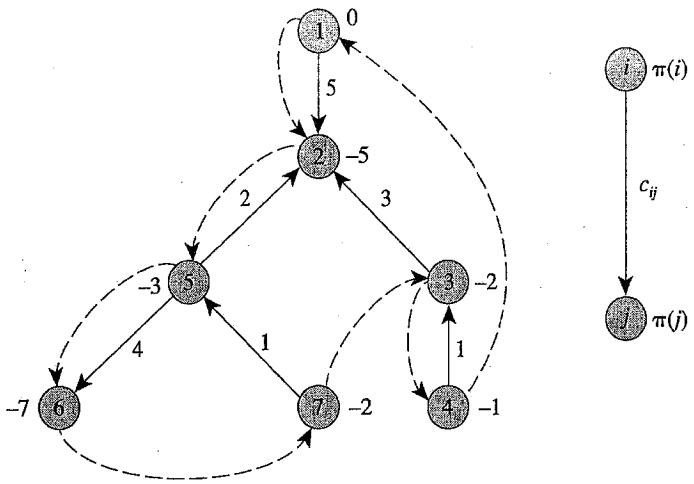


Figure 11.5 Computing node potentials for a spanning tree.

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.

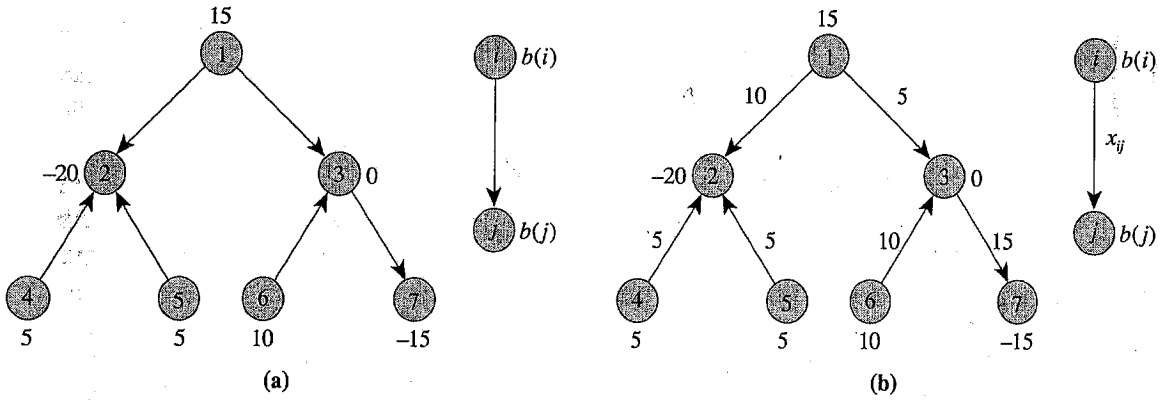


Figure 11.6 Computing flows for a spanning tree.

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 **U** requires $O(m)$ time. If we set aside the time to select leaf nodes of **T**, then each iteration requires $O(1)$ time, resulting in a total of $O(n)$ time. One way of identifying leaf nodes in **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 $0 \le x_{ij} \le u_{ij}$ 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 $(i, j) \in$ 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 $b(j) \ge 0$, 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 $(i, j) \in$ 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 $-c_{kl}^{\pi}$; if $l \in T_1$ and $k \in T_2$, they increase by the amount c_{kl}^{π}. 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)

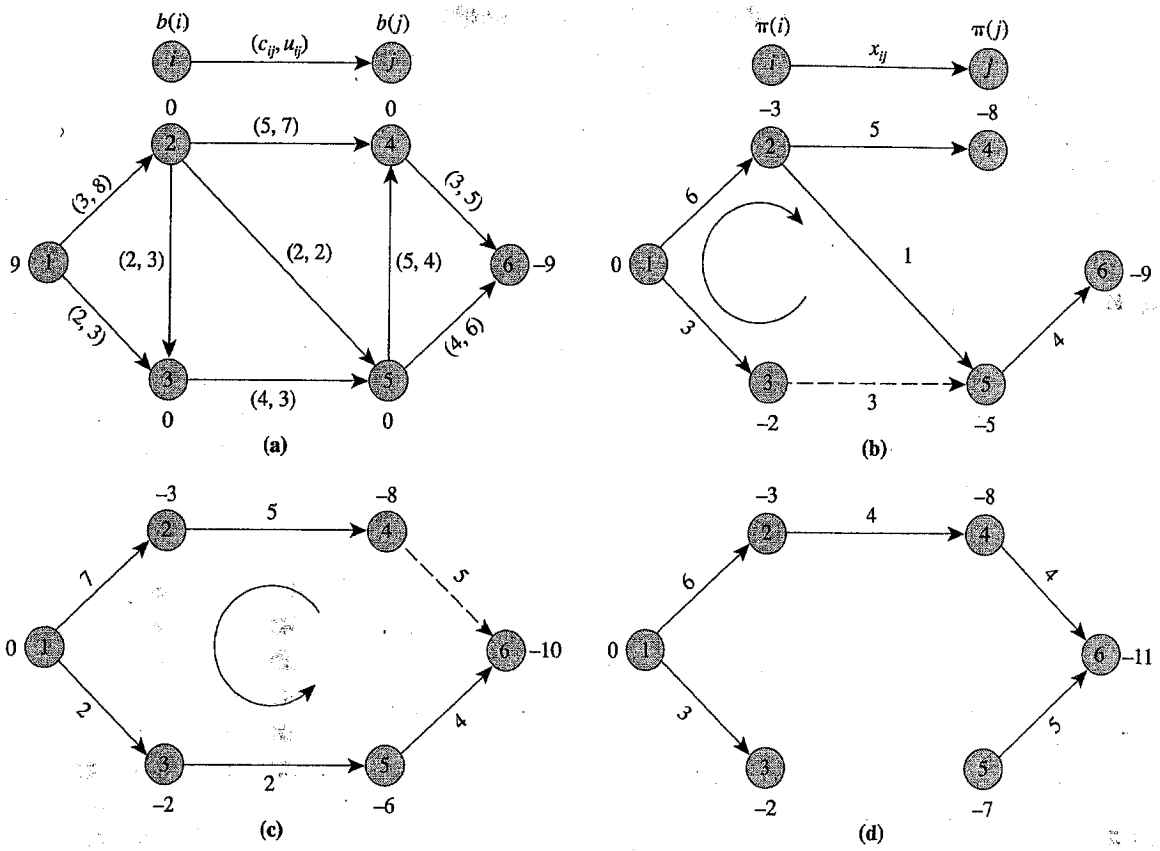


Figure 11.11 Numerical example for the network simplex algorithm.

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

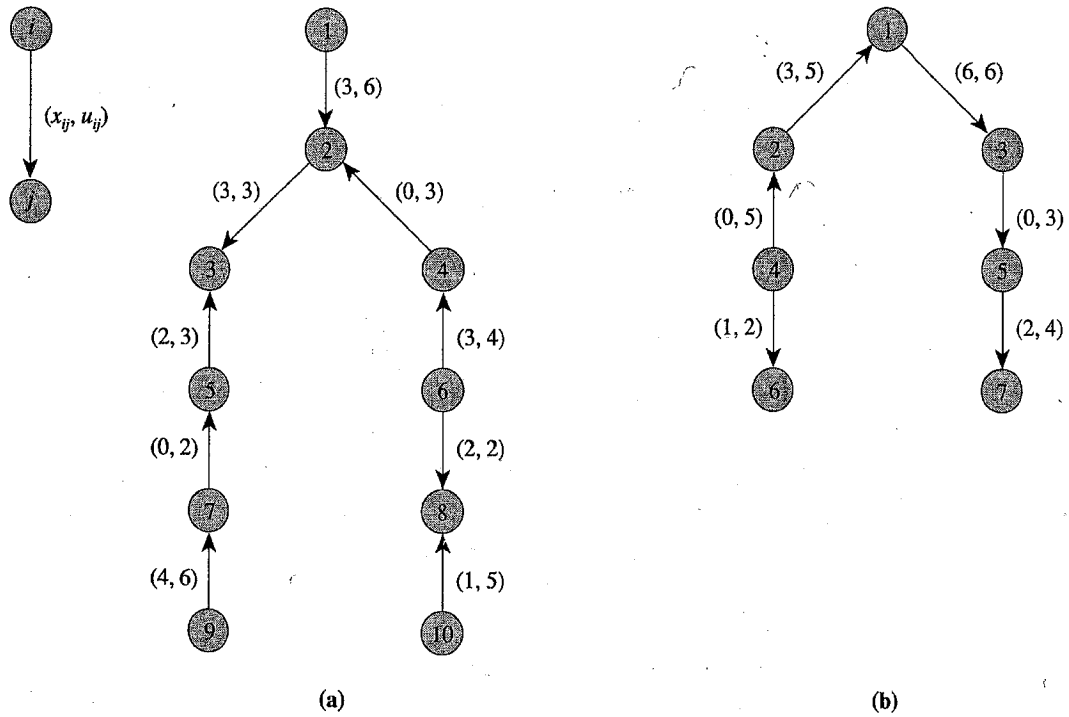


Figure 11.12 Feasible spanning trees: (a) strongly feasible; (b) nonstrongly feasible.

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

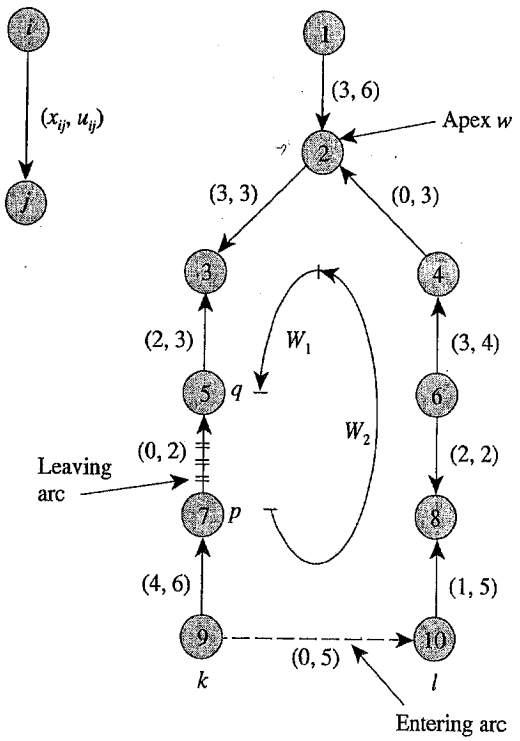


Figure 11.13 Selecting the leaving arc.

(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

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

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

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.

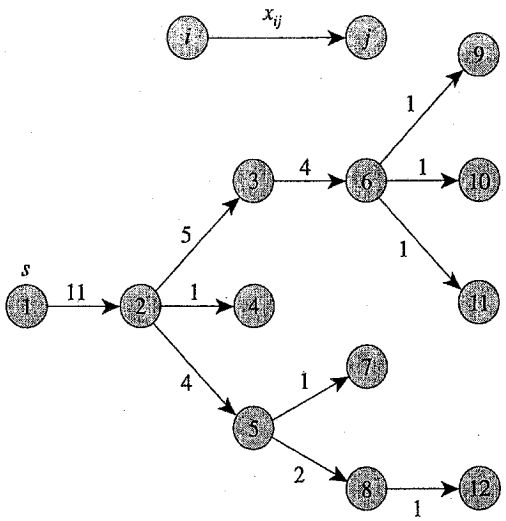


Figure 11.14 Directed out-tree rooted at the source.

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

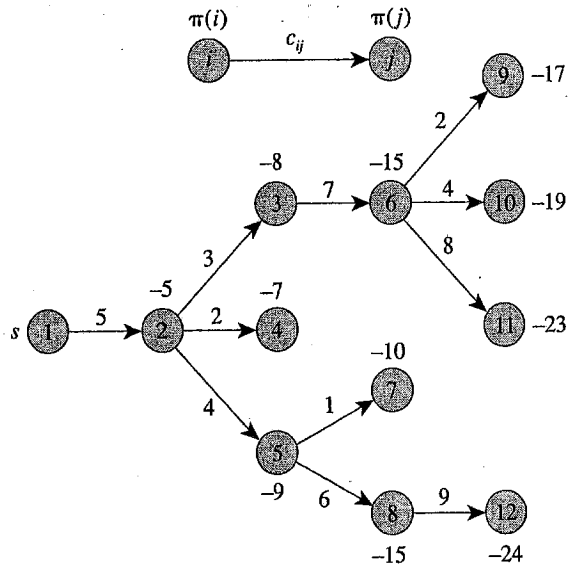


Figure 11.15 Computing node potentials.

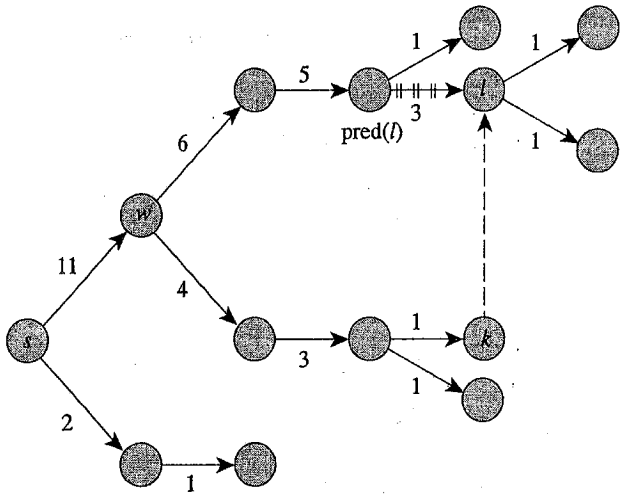


Figure 11.16 Selecting the leaving arc.

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.

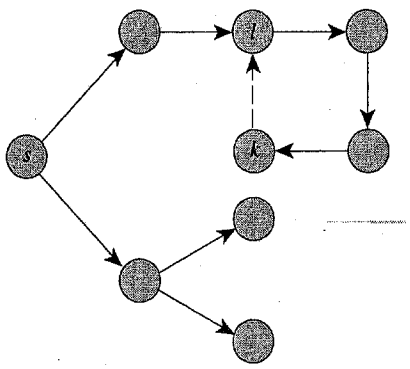


Figure 11.17 Detecting a negative cycle in the network.

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, $-nC \le \pi(i) \le nC$ (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 $\pi^*(p) - \pi(p) \le 2n\Delta$ for each node p.*

Proof. In the first scaling phase, $\Delta \ge C$ 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), $c_{ij}^{\pi} \ge -\Delta$ 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.

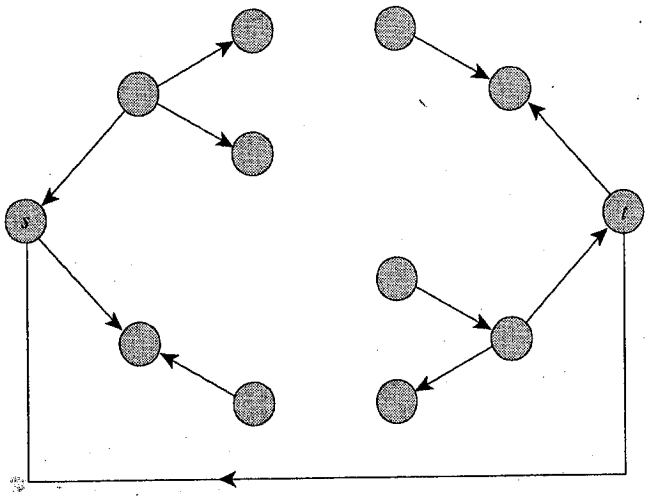


Figure 11.18 Spanning tree for the maximum flow problem.

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.

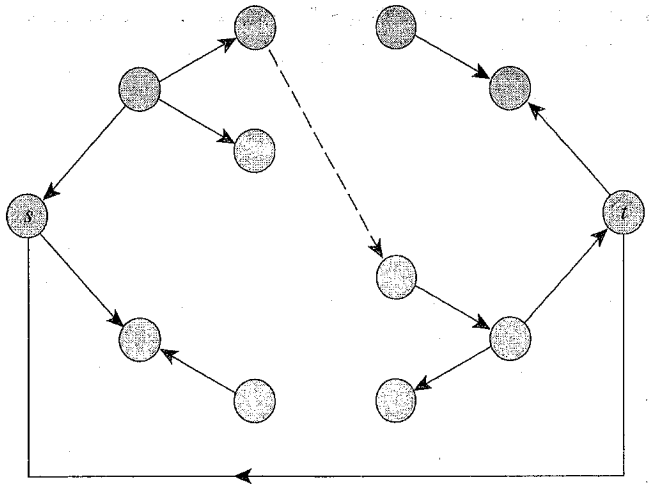


Figure 11.19 Forming a cycle.

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 $(\mathbf{T}, \mathbf{L}, \mathbf{U})$ 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

$$\theta_{kl} \leq \theta_{ij} \text{ for each } (i, j) \in ((S, \bar{S}) \cap \mathrm{L}) \cup ((\bar{S}, S) \cap \mathrm{U}) \doteq Q.$$

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

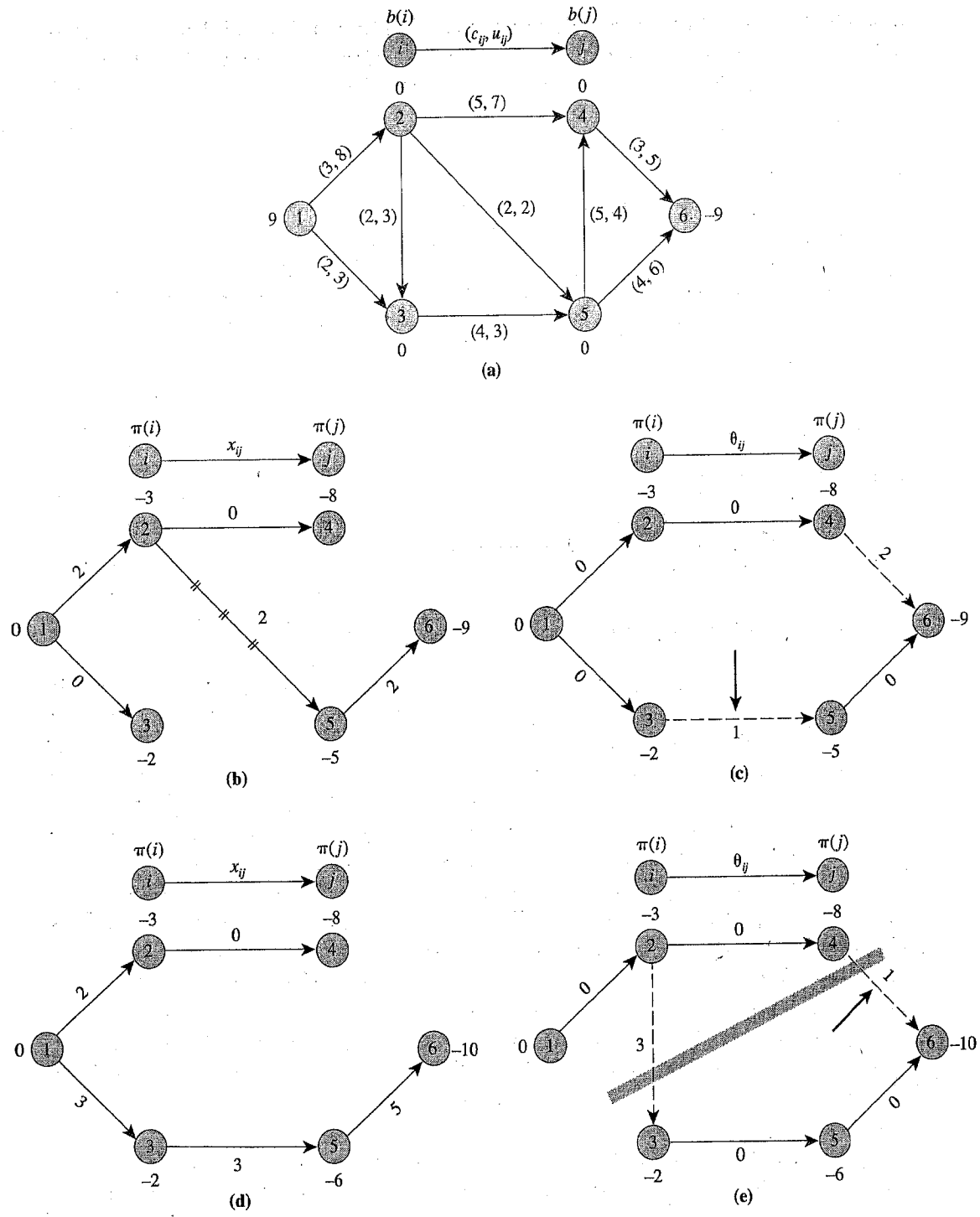


Figure 11.20 Illustrating the parametric network simplex algorithm.

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

$$Q = ((S, \bar{S}) \cap \mathbf{L}) \cup ((\bar{S}, S) \cap \mathbf{U}).$$

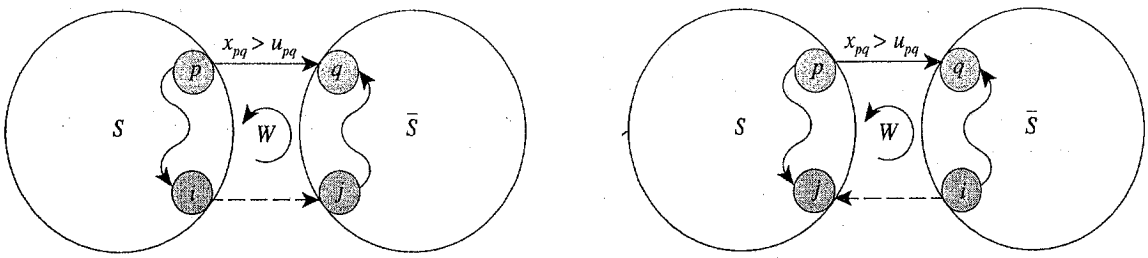


Figure 11.21 Identifying eligible arcs in the dual network simplex algorithm.

If $Q = \emptyset$, we cannot reduce the flow on arc (p, q) and the minimum cost flow problem is infeasible (see Exercise 11.37). If $Q \neq \emptyset$, 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

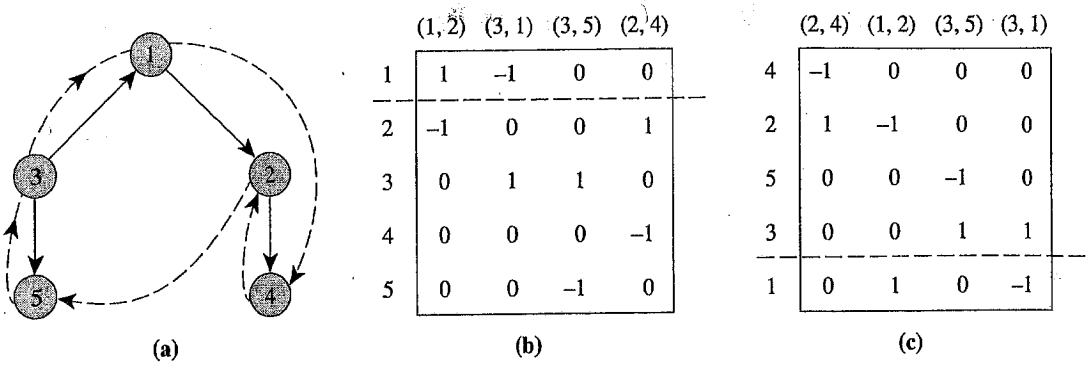


Figure 11.22 (a) Spanning tree and its reverse thread traversal; (b) basis matrix corresponding to the spanning tree; (c) basis matrix after rearranging the rows and columns.

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 **L** ∪ **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, $\overline{\mathcal{N}}_{kl}$, 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 N of a directed network is totally unimodular.*

Proof. To prove the theorem, we need to show that every square submatrix $\mathcal{F}$ of N of size k has determinant 0, $+1$, or -1. We establish this result by performing induction on k. Since each element of 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 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.

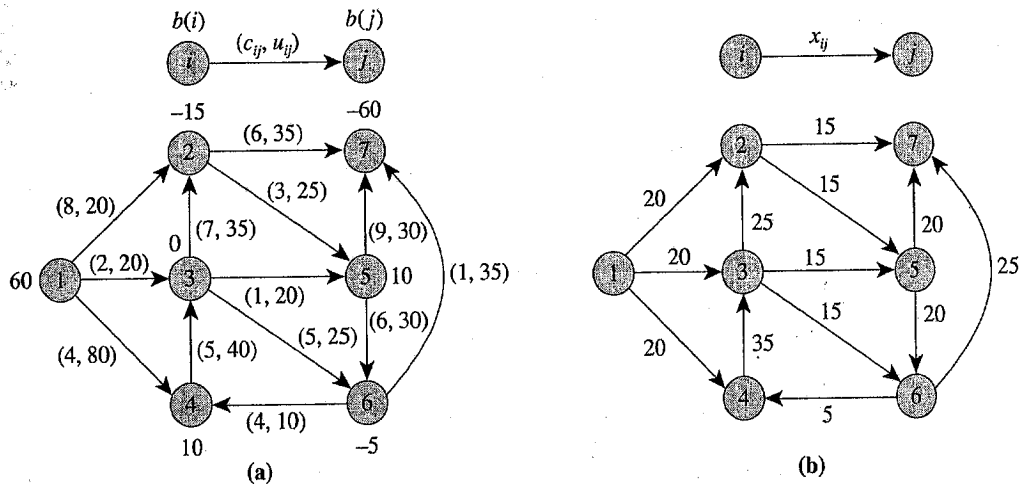


Figure 11.24 Example for Exercise 11.12: (a) problem data; (b) feasible solution.

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.

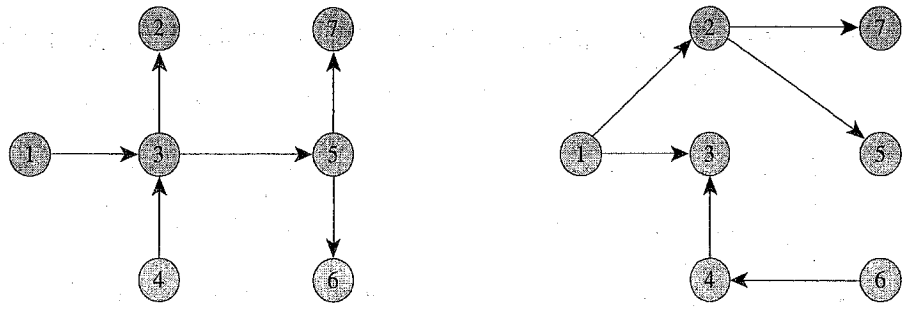


Figure 11.25 Two spanning trees of the network in Figure 11.24.

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 $\mathbf{U} = \{(1, 3), (2, 4), (5, 6)\}$.

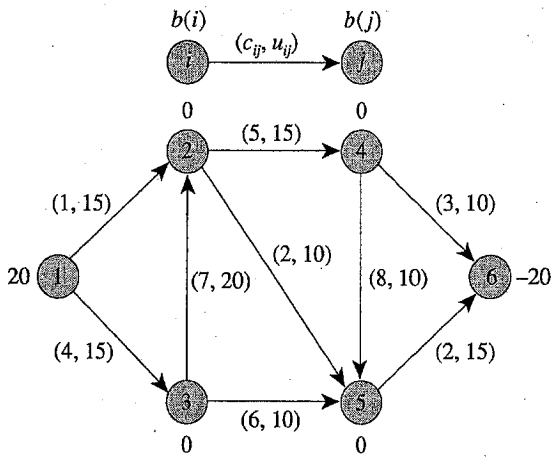


Figure 11.26 Example for Exercises 11.16 and 11.17.

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 **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 **T** be a spanning tree of G and let $D(j)$ denote the set of descendants of node j in **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 (**T**, **L**, **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 (**T**, **L**, **U**) is strongly feasible if and only if (**T**, **L**, **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.)

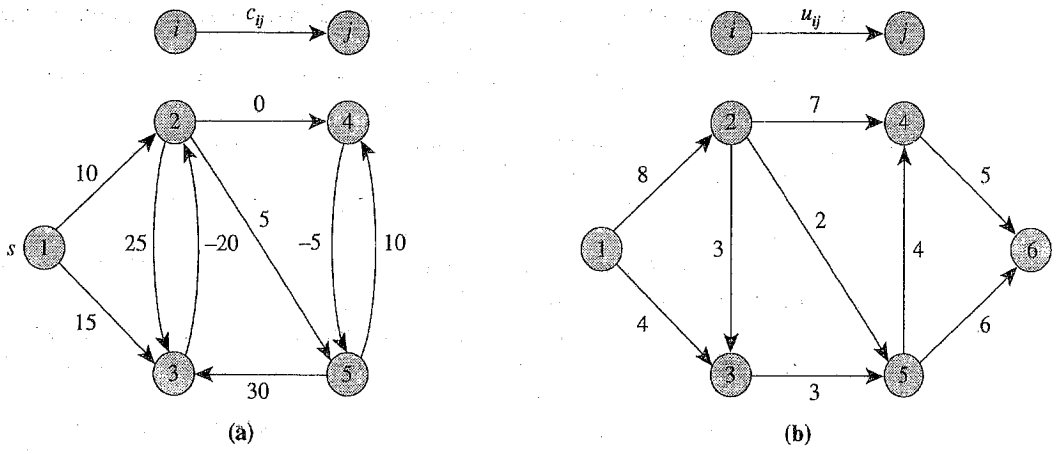


Figure 11.27 Examples for Exercises 11.28 and 11.29.

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 $\mathcal{N}^T$ 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 T (we can compute these potentials using the procedure *compute-potentials*). Show that $\pi^0 + \lambda\pi^*$ are the node potentials for the tree 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 (T, L, 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 (T, L, 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 (T, L, U) be the basic structure at some state. Maintain the flow on each arc in the set 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 $\mathbf{T} \cup \{(p, q)\}$, L, and U satisfies the following two properties: (1) 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.

12

ASSIGNMENTS AND MATCHINGS

It takes two to tango.
—From a popular American song

Chapter Outline

12.1 INTRODUCTION

To this point in our discussion, we have focused on the three major building blocks of network flows: shortest paths, maximum flows (or minimum cuts), and minimum cost flows. We have seen how these models arise in numerous application settings, we have studied a number of different solution strategies and specific algorithms for solving these problems, and we have seen how important data structures can be used in designing algorithms and in implementing them efficiently. Many of these same ideas apply more generally to the broader field of combinatorial (or discrete) optimization and, indeed, many results in this broader field build on those developed for network flows. This chapter, which considers a particular class of combinatorial optimization problems known as *matching problems*, illustrates the flow of ideas from network flows to other arenas of discrete optimization.

In general, in discrete optimization we are given a finite set of objects and an objective function defined on these objects, and we wish to choose the object with the smallest (or largest) objective value. In this most general form, discrete optimization problems are hopelessly difficult to solve unless we impose some structure on the finite set and on the objective function. In fact, we might view the field of combinatorial optimization as the study of those structures that permit us to say something interesting about the nature of the underlying optimization problem or permit us to find an optimal solution efficiently.

Network flow problems certainly define one very important class of specially structured combinatorial optimization problems. With the exception of our treatment of the combinatorial implications of the max-flow min-cut theorem in Chapter 6, we have not emphasized this viewpoint very much in our discussion. Nevertheless, all

of the problems we have considered fit this description of discrete optimization problems. For example, in the context of the shortest path problem, the finite objects are all the paths joining two nodes in a network and the objective function is additive over the arcs selected in any path. For the minimum cut problem, the finite set is the set of cuts separating the source and the sink and the objective function is again additive over the arcs. For the maximum flow problem with unit arc capacities, the finite objects are all sets of paths from the source to the sink that are arc-disjoint and we wish to find the solution with the largest number of paths (by replacing arcs with integer capacities by parallel arcs with unit capacities, we can interpret all maximum flow problems in a similar fashion). For minimum cost flow problems with unit supplies and demands at the nodes, the underlying objects are the set of paths directed from a supply node to a demand node; the objective function in this case is the sum of the weight of arcs in the chosen paths. (By duplicating nodes with integral supplies and demands into sets of nodes with unit supplies and demands, we can interpret more general minimum cost flow problems in this same way.)

In this and the next chapter, we consider two related combinatorial models that are defined over graphs with a weight associated with each arc. In Chapter 13 the objects are all the spanning trees in the network and we wish to find the spanning tree with the smallest overall weight (defined as the sum of the weights of its constituent arcs). This problem is known as the *minimum spanning tree problem*. In this chapter the objects are all subgraphs, called *matchings*, with the property that every node in the subgraph has degree zero or one. That is, no two arcs in the subgraph are incident to the same node. We wish to find the matching with the smallest overall weight, again defined as the sum of the weights of its constituent arcs.

The matching problem arises in many different problem settings since we often wish to find the best way to pair objects or people together to achieve some desired goal. The classical bipartite matching problem is a special case in which objects separate into two groups, and we wish to pair the objects in the different groups in some optimal fashion—for example, we wish to assign jobs to machines in the most cost-effective manner. The general matching problem models situations in which the objects need not fall into two groups—that is, the underlying network need not be bipartite.

We begin this chapter by describing several applications of matching problems in practical contexts as varied as inventory planning, machine scheduling, drilling oil fields, and personnel assignment. We then describe solution approaches for several special cases of the general matching problem. We begin by examining bipartite matching problems. We consider two versions of these problems: (1) the cardinality problem in which we wish to find a matching containing the maximum number of arcs, and (2) the weighted problem in which we have a weight associated with each arc and we wish to find a matching with the largest overall weight (for the cardinality problem, the weights are all 1). For the weighted problem, we restrict the matching to a smaller class of subgraphs, known as perfect matchings, in which every node is incident to exactly one arc in the matching (i.e., every node has degree exactly 1).

As we will see, bipartite problems are easy to solve because we can model them as network flow problems and solve them using any of the many algorithms

that we have already studied. Because the resulting network flow problems have a special structure, we can refine our analysis of the network flow algorithms and show that their worst-case complexity is better for the bipartite matching problems than they are in general. In particular, we show that maximum flow algorithms solve the cardinality bipartite matching problem in $O(\sqrt{n}m)$ time. We also show that if $S(n, m, C)$ denotes the time required to solve a shortest path problem on an n-node and m-arc network with nonnegative arc costs bounded by C, specializations of the successive shortest path algorithm and the primal–dual minimum cost flow algorithms solve the weighted bipartite matching problem in $O(n\,S(n, m, C))$ time. We also describe a cost scaling algorithm with an even better time bound.

Nonbipartite matching problems are more difficult to solve because they do not reduce to standard network flow problems. Therefore, they require specialized combinatorial algorithms. To demonstrate the flavor of these algorithms but not go too far afield from the general thrust of this book, we consider only the cardinality version of the nonbipartite matching problem. We describe a clever $O(n^3)$ augmenting path algorithm for solving this problem. Even though the details of this algorithm are quite different from those of the algorithms we have studied for solving core network flow problems, the algorithm does adopt a common algorithmic strategy that we have seen many times before in previous chapters.

In this chapter we also consider a variant of the matching problem known as the stable marriage problem. This model differs from other models we have considered in the text in one important respect: It has no objective function that we wish to optimize. Instead, it models situations with two groups, such as men and women, in which each man has a ranking of each woman and each woman has a ranking of the men. We seek a feasible matching of the members of the two groups, known as a *stable matching*, with the property that no pair of man and woman prefer each other to the partners that they have in the stable matching. We show that for any set of rankings, this problem always has a stable matching and we show how to compute such a solution in $O(n^2)$ time.

12.2 APPLICATIONS

As we show in this section, matching problems arise in a variety of different problem contexts. In Chapter 1 we considered two applications, the pairing of stereo speakers to achieve balanced frequency responses and the rewiring of typewriters. We now describe several other applications.

Application 12.1 Bipartite Personnel Assignment

In many different problem contexts, we wish to assign people to objects: for example, to jobs, machines, rooms, or each other. Each assignment has a "value" and we wish to make the assignments so that we maximize the sum of these values. To illustrate the range of these contexts, in this and the next application, we consider six different applications relating to personnel assignment.

1. A firm has hired n graduates to fill n vacant jobs. Based on aptitude tests, college grades, and letters of recommendation, the firm has assigned a *profi-*

ciency index u_{ij} for placing candidate i in job j. The objective is to identify an assignment that maximizes the total proficiency score over all jobs. This problem is clearly an application of the assignment problem.

2. A swimming coach must select from his eight best swimmers a medley relay team of four, each of whom will then swim one of the four strokes (back, breast, butterfly, and free-style). The coach knows the time of each swimmer in each stroke. The problem is to identify the team of the four best swimmers out of the eight that are available. Clearly, the sum of times obtained by optimally matching four out of the eight swimmers to the four strokes gives the minimum feasible relay time and the corresponding team is the best team. We point out that in this version of the assignment problem $| N_1 | > | N_2 |$; nevertheless, by adding "dummy nodes," we can easily transform this problem into an equivalent one in which both node sets N_1 and N_2 have the same size.
3. In the armed forces, many men and women are qualified to perform specific jobs, or postings. The armed forces would like to assign the service personnel to postings in order to minimize moving costs. General rules specify the needed qualifications of the personnel for the postings and identify jobs that need to be filled. Policy rules determine allowable assignments that reflect job qualifications and personnel requirements. For an allowable assignment, the *posting cost* is the dollar cost of moving the person, his or her family, and his or her belongings to the new residence. In this case the assignment problem would find an allowable assignment that minimizes the total posting cost.

Application 12.2 Nonbipartite Personnel Assignment

1. During World War II, the Royal Air Force (RAF) of Britain contained many pilots from foreign countries who spoke different languages and had different levels of training. The RAF had to assign two pilots to each plane, always assigning pilots with compatible languages and training to the same plane. The RAF wanted to fly as many planes as possible. To formulate this problem as a maximum cardinality matching problem, we define a graph whose nodes represent pilots; we join two nodes by an arc if the corresponding pilots are compatible.
2. A hostel manager wants to assign pairs of roommates to rooms of her hostel. The nationality, religion, cultural background, and hobbies determine compatible pairs of roommates. So the problem of finding the maximum number of compatible pairs is a maximum cardinality matching problem.
3. Suppose that an airline wishes to divide its $2p$ airplane pilots, linearly ordered by seniority (with no ties), into m teams each containing a captain and a first officer. The captain of each team must have seniority over the first officer. Each pilot i has a measure, α_i, of his effectiveness as a captain and another, β_i, measure of his effectiveness as a first officer. We seek an assignment of pilots to teams that will maximize the total measure of effectiveness summed over all the teams. This problem is an instance of the maximum weight matching problem: We represent each pilot as a node and define the cost of an arc (i, j)

as $\alpha_j + \beta_i$ if pilot j is more senior than pilot i, and as $\alpha_i + \beta_j$ if pilot i is more senior than pilot j.

Application 12.3 Assigning Medical School Graduates to Hospitals

Each year medical schools in the United States graduate thousands of doctors who are eligible for residencies at the various hospitals across the country. To give each of the graduates a chance to find the "best possible" residency and the hospitals the chance to obtain the "best possible" residents, the American Medical Association (AMA) conducts a matching process in which the graduates rank the hospitals according to their preferences and the hospitals rank the graduates according to their preferences. It then assigns the graduates to hospitals so that the matching is "stable" in the following sense. We say that an assignment is *unstable* if some graduate i is not assigned a hospital j, but that graduate prefers hospital j over his or her current assignment and, at the same time, hospital j prefers graduate i over one of the graduates assigned to it. This assignment is unstable because both the graduate i and the hospital j have an incentive to change their current assignments. We refer to an assignment that is not unstable as *stable*. The objective of the AMA is to identify a stable assignment. This problem is an example of the stable marriage problem that we discuss in Section 12.5.

Application 12.4 Dual Completion of Oil Wells

An oil company has identified several individual oil traps, called *targets*, in an offshore oil field and wishes to drill wells to extract oil from these traps. Figure 12.1 illustrates a situation with eight targets. The company can extract any target separately (so-called *single completion*) or extract oil from any two targets together by drilling a single hole (so-called *dual completion*). It can estimate the cost of drilling and completing any target as a single completion or any pair of targets as a dual completion. This cost will depend on the three-dimensional spatial relationships of targets to the drilling platform and to each other. The decision problem is to determine which targets (if any) to drill as single completions and which pairs to drill together as duals, so as to minimize the total drilling and completion costs. If we restrict the solution to use only dual completions, the decision problem is a nonbipartite weighted matching problem.

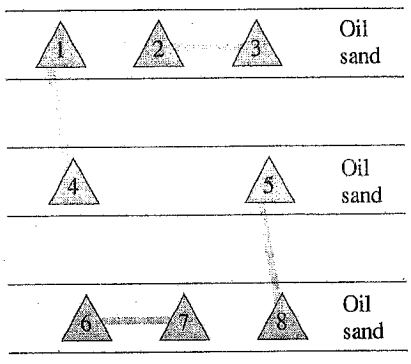


Figure 12.1 Targets and matchings for the dual completion problem.

Application 12.5 Determining Chemical Bonds

Matching problems arise in the field of chemistry as chemists attempt to determine the possible atomic structures of various molecules. Figure 12.2(a) specifies the partial chemical structure of a molecule of some hydrocarbon compound. The molecule contains carbon atoms (denoted by nodes with the letter "C" next to them) and hydrogen atoms (denoted by nodes with the letter "H" next to them). Arcs denote bonds between atoms. The bonds between the atoms, which can be either single or double bonds, must satisfy the "valency requirements" of all the nodes. (The valency of an atom is the sum of its bonds.) Carbon atoms must have a valency of 4 and hydrogen atoms a valency of 1.

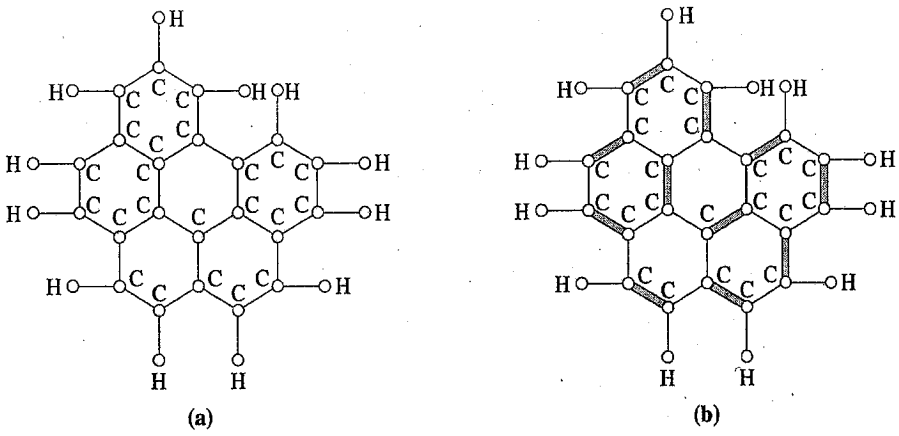


Figure 12.2 Determining the chemical structure of a hydrocarbon.

In the partial structure shown in Figure 12.2, each arc depicts a single bond and, consequently, each hydrogen atom has a valency of 1, but each carbon atom has a valency of only 3. We would like to determine which pairs of carbon atoms to connect by a double bond so that each carbon atom has valency 4. We can formulate this problem of determining some feasible structure of double bonds as an instance of a perfect matching problem in the network obtained by deleting the hydrogen atoms and those carbon atoms with valency 4. Figure 12.2(b) gives one feasible bonding structure of the compound; the bold lines in this network denote double bonds between the atoms.

Application 12.6 Locating Objects in Space

To identify an object in (three-dimensional) space, we could use two infrared sensors, located at geographically different sites. Each sensor provides an angle of sight of the object and hence the line on which the object must lie. The unique intersection of the two lines provided by the two sensors (provided that the two sensors and the object are not collinear) determines the unique location of the object in space.

Consider now the situation in which we wish to determine the locations of p objects using two sensors. The first sensor would provide us with a set of lines L_1, $L_2, \ldots, L_p$ for the p objects and the second sensor would provide us a different

set of lines $L_1', L_2', \ldots, L_p'$. To identify the location of the objects—using the fact that if two lines correspond to the same object, the lines intersect one another—we need to match the lines from the first sensor to the lines from the second sensor. In practice, two difficulties limit the use of this approach. First, a line from a sensor might intersect more than one line from the other sensor, so the matching is not unique. Second, two lines corresponding to the same object might not intersect because the sensors make measurement errors in determining the angle of sight. We can overcome this difficulty in most situations by formulating this problem as an assignment problem.

In the assignment problem, we wish to match the p lines from the first sensor with the p lines from the second sensor. We define the cost c_{ij} of the assignment (i, j) as the minimum Euclidean distance between the lines L_i and L_j. We can determine c_{ij} using standard calculations from geometry. If the lines L_i and L_j correspond to the same object, c_{ij} would be close to zero. An optimal solution of the assignment problem would provide an excellent matching of the lines. Simulation studies have found that in most circumstances, the matching produced by the assignment problem defines the correct location of the objects.

Application 12.7 Matching Moving Objects

In several different application contexts, we might wish to estimate the speeds and the directions of movement of a set of p objects (e.g., enemy fighter planes, missiles) that are moving in space. Using the method described in the preceding application, we can determine the location of the objects at any point in time. One plausible way to estimate the objects' movement directions and speeds is to take two snapshots of the objects at two distinct times and then to match one set of points with the other set of points. If we match the points correctly, we can assess the speed and direction of movement of the objects. As an example, consider Figure 12.3 which denotes the objects at time 1 by squares and the objects at time 2 by circles.

Let (x_i, y_i, z_i) denote the coordinates of object i at time 1 and (x_i', y_i', z_i') denote the coordinates of the same object at time 2. We could match one set of points with

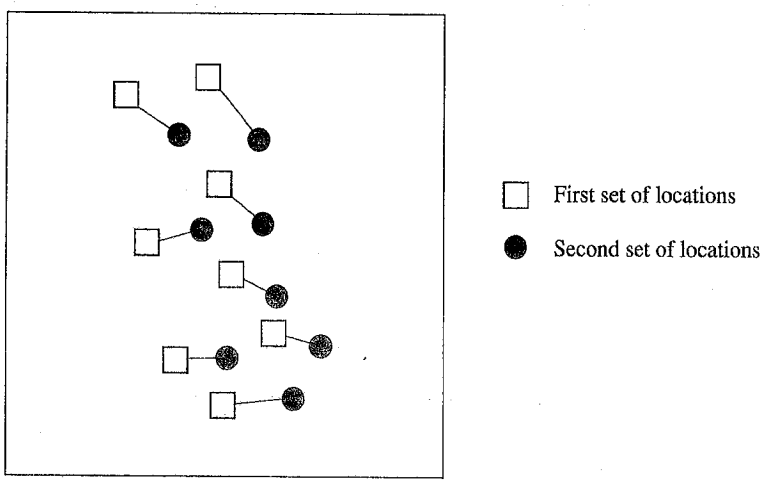


Figure 12.3 Two snapshots of a set of eight objects.

the other set of points in many ways. Minimizing the sum of the squared Euclidean distances between the matched points is quite appropriate in this scenario because it attaches a higher penalty to larger distances. If we take the snapshots of the objects at two times that are sufficiently close to each other, the optimal assignment will often match the points correctly. In this application of the assignment problem, we let $N_1 = \{1, 2, \ldots, p\}$ denote the set of objects at time 1, let $N_2 = \{1', 2', \ldots, p'\}$ denote the set of objects at time 2, and define the cost of an arc (i, j') as $[(x_i - x_j')^2 + (y_i - y_j')^2 + (z_i - z_j')^2]$. The optimal assignment in this graph will specify the desired matching of the points. From this matching we obtain an estimate of the movement directions and the velocities of the individual objects.

Application 12.8 Optimal Depletion of Inventory

In many different problem contexts, we need to store items that either deteriorate or increase in value over time. Suppose that we have a stockpile consisting of p items of the same type. Item i has a current age a_i. A function $v(t)$ specifies the expected utility (or value) for an item of age t when we withdraw it from the stockpile. We need to meet a given schedule that specifies the times at which items are required. The problem is to determine the order for issuing the items that maximizes the total expected utility summed over all p items. A specific example of this general problem is the storage of a number of vats of a volatile liquid (e.g., alcohol). Since alcohol is volatile, we incur an increasing loss due to evaporation with age in storage, so the value of the vat decreases over time.

Some special instances of this inventory problem are particularly easy to solve; for example, when the utility function $v(t)$ satisfies convexity or concavity properties (see Exercise 12.5). When the utility function is arbitrary, we can solve the problem as an assignment problem. Let $t_1, t_2, \ldots, t_p$ denote the time instances when we need to extract an item from the stockpile. Then since item i has an age a_i at time zero, the expected utility for the issue of ith item at time t_j is

$$u_{ij} = v(a_i + t_j).$$

If we compute these utilities for all pairs of i and j, we solve the inventory problem by solving the $p \times p$ assignment problem of maximizing the assignment utilities.

Application 12.9 Scheduling on Parallel Machines

In many application settings, such as the scheduling of computer programs on processors of a computer, we are given a set of w jobs each requiring processing on one of r machines. Suppose that job i requires a processing time of p_{ij} on machine j. Our aim is to find an assignment of the jobs to the machines and a machine schedule (i.e., an order for performing the jobs assigned to the same machine) that will minimize the total flow time of jobs. The *flow time* of a job is the time the job spends in the system before the machines have completed its processing. For example, if we assign jobs 1, 4, and 5 to machine 2 in the order 4–1–5, the flow time of job 4 is p_{24}, the flow time of job 1 is $p_{24} + p_{21}$, and the flow time of job 5 is $p_{24} + p_{21} + p_{25}$. Consequently, the total flow times of the jobs assigned to machine 2 is $1(p_{25})$

$+ 2(p_{21}) + 3(p_{24})$. Observe that to determine the total flow time of jobs allocated to a specific machine, we multiply the processing time of the last job by 1, the processing time of the second to last job by 2, and so on, and sum these numbers.

Any algorithm for this scheduling problem must accomplish two objectives. First, it must assign jobs to the various machines. Second, it must sequence the processing of the jobs assigned to any single machine (i.e., assign one job to the first place, one job to the second place, etc.). We would like to make these assignments to minimize the total flow time. This viewpoint suggests that we can assign a job j in one of the wr ways: We can assign it to one of the r machines i (so i can vary from 1 to r) and to one of the kth to last positions on this machine (so k can vary from 1 to w). The cost of this specific assignment would be kp_{ij}.

This scheduling problem is an assignment problem on a network $G = (N_1 \cup N_2, A)$ with nodes N_1 representing the jobs (so $| N_1 | = w$) and with nodes N_2 representing the places on different machines (so $| N_2 | = wr$). Each node in N_1 in this network is connected to every node in N_2 and the cost of any arc is the cost of assigning a job to a specific place on any machine. In Exercise 12.6 we provide a more rigorous set of arguments for establishing the validity of this formulation.

At first glance, the resulting assignment problem might appear to be much larger than the scheduling problem. However, it is possible to obtain a bound on the maximum number of jobs assigned to the machines and thus to reduce the size of the assignment problem substantially for most instances of the scheduling problem.

12.3 BIPARTITE CARDINALITY MATCHING PROBLEM

As defined earlier, in the bipartite cardinality matching problem (or simply the bipartite matching problem), we wish to identify a matching of maximum cardinality in a bipartite undirected network. Several efficient algorithms for solving this problem achieve a worst-case bound of $O(\sqrt{n}m)$. One approach is to transform the problem into a maximum flow problem in a simple network. In this section we study this approach. We discuss another approach in Section 12.7 as a stepping stone for developing an algorithm for the nonbipartite cardinality matching problem.

Recall from Section 8.2 that in a simple network, each arc has a unit capacity and each node has an indegree of at most 1 or an outdegree of at most 1. To transform a bipartite matching problem defined on an undirected graph $G = (N_1 \cup N_2, A)$ into a maximum flow problem, we first create a directed version of the underlying graph G by designating all arcs as pointing from the nodes in N_1 to the nodes in N_2. We then introduce a source node s and a sink node t, with an arc connecting s to each node in N_1 and an arc connecting each node in N_2 to t. We set the capacity of each arc in the network to 1. Figure 12.4 illustrates this transformation. We refer to the transformed network as $G' = (N', A')$. Note that the network G' is a simple network since every node in N_1 has one incoming arc and every node in N_2 has one outgoing arc. We now establish a one-to-one correspondence between a matching of cardinality k in the original network and an integral flow of value k in the transformed network.

Given a matching $\{(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)\}$ of cardinality k in the original network G, we construct a flow in the transformed network G' as follows. We first set the flow on each of the matched arcs equal to 1. Then to satisfy the mass balance

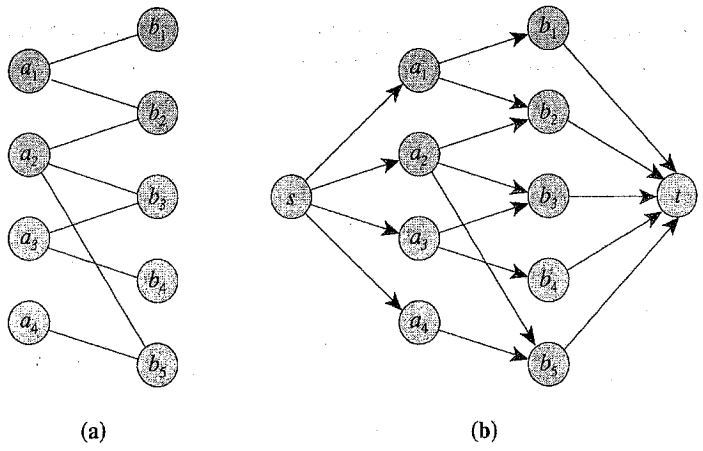


Figure 12.4 Transforming a bipartite (cardinality) matching problem to a maximum flow problem: (a) original network; (b) unit capacity maximum flow network.

constraints, we set the flow on the arcs (s, i_r) and (j_r, t) equal to 1 for all $r = 1, 2, \ldots, k$. Clearly, this choice gives us a flow of value k from node s to node t.

Similarly, given an integral flow of value k from node s to node t in the transformed network, we can specify a corresponding matching in the original network: By flow decomposition, the integral flow of cardinality k decomposes into k paths of the form $s - i_1 - j_1 - t, s - i_2 - j_2 - t, \ldots, s - i_k - j_k - t$. Since each of the arcs incident to nodes s and t have a unit capacity, no two nodes in N_1 or N_2 appear in more than one of these paths, and so the k arcs $\{(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)\}$ define a matching.

We have thus established an equivalence between matchings in the original network and integral flows in the transformed network. Therefore, to solve the matching problem, we solve a maximum flow problem in the transformed network using the $O(\sqrt{n}m)$ time algorithm described in Section 8.2. Recall that this algorithm produces an integer optimal flow. The matching corresponding to the maximum flows is a maximum cardinality matching. We have therefore established the following result.

Theorem 12.1. *It is possible to solve the maximum cardinality bipartite matching problem in $O(\sqrt{n}m)$ time.*

12.4 BIPARTITE WEIGHTED MATCHING PROBLEM

In this section we study the bipartite weighted matching problem; namely, given a weighted bipartite network $G = (N_1 \cup N_2, A)$ with $|N_1| = |N_2|$ and arc weights c_{ij}, find a perfect matching of minimum weight. We allow the network G to be directed or undirected. If the network is directed, we require that for each arc $(i, j) \in A$, $i \in N_1$ and $j \in N_2$. If the network is undirected, we make it directed by designating all arcs as pointing from the nodes in N_1 to those in N_2. We shall, therefore, henceforth assume that G is a directed graph. In the operations research literature, the bipartite weighted matching problem is known as the *assignment problem*; for the sake of brevity and to conform with this convention, we adopt this terminology.

Recall that the assignment problem is a special case of the minimum cost flow problem and can be stated as the following linear program.

$$\text{Minimize} \quad \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{12.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} = 1 \qquad \text{for all } i \in N_1, \tag{12.1b}$$

$$\sum_{\{j:(j,i)\in A\}} x_{ji} = 1 \qquad \text{for all } i \in N_2, \tag{12.1c}$$

$$x_{ij} \geq 0 \qquad \text{for all } (i, j) \in A. \tag{12.1d}$$

Since we can formulate the weighted bipartite matching problem as this special type of flow problem, it is not too surprising to learn that most algorithms for the assignment problem can be viewed as adaptations of algorithms for the minimum cost flow problem. However, the special structure of the assignment problem often permits us to simplify these algorithms and to obtain improved bounds on their running times.

One popular algorithm for the assignment problem is a specialization of the network simplex algorithm discussed in Chapter 11. Another popular algorithm is the successive shortest path algorithm and its many variants. In the following discussion, we briefly describe some of these successive shortest path-based algorithms. We also describe an adaptation of the cost scaling algorithm.

Successive Shortest Path Algorithm

This algorithm is a direct implementation of the successive shortest path algorithm for the minimum cost flow problem discussed in Section 9.7. Recall that the successive shortest path algorithm obtains shortest path distances from a supply node to all other nodes in a residual network, uses these distances to update node potentials and then augments flow from that supply node to a demand node. This algorithm, when applied to the assignment problem, would augment 1 unit flow in every iteration, which would amount to assigning one additional node in N_1. Consequently, if we let $S(n, m, C)$ denote the time needed to solve a shortest path problem with nonnegative arc lengths and let $n_1 = |N_1|$, the algorithm would terminate within n_1 iterations and would require $O(n_1 S(n, m, C))$ time.

Hungarian Algorithm

The Hungarian algorithm is a direct implementation of the primal–dual algorithm for the minimum cost flow problem that we discussed in Section 9.8. Recall that the primal–dual algorithm first transforms the minimum cost flow problem into a problem with a single supply node s^* and a single demand node t^*. At every iteration, the primal–dual algorithm computes shortest path distances from s^* to all other nodes, updates node potentials, and then solves a maximum flow problem that sends the maximum possible flow from node s^* to node t^* over arcs with zero reduced costs. When applied to the assignment problem, this algorithm terminates within n_1 iter-

ations since each iteration sends at least 1 unit of flow, and hence assigns at least one additional node in N_1. The time required to solve shortest path problems in all these iterations is $O(n_1 S(n, m, C))$. Next consider the total time required to establish maximum flows. The labeling algorithm, described in Section 6.5, for solving the maximum flow problem would require a total of $O(nm)$ time because it would perform n augmentations and each augmentation requires $O(m)$ time. The dominant portion of these computations is the time required to solve shortest path problems. Consequently, the overall running time of the algorithm is $O(n_1 S(n, m, C))$.

Relaxation Algorithm

The relaxation algorithm, which is closely related to the successive shortest path algorithm, is another popular approach for solving the assignment problem. This algorithm relaxes the constraint (12.1c), thus allowing any node in N_2 to be assigned to more than one node in N_1. The relaxed problem is easy to solve: We assign each node $i \in N_1$ to any node $j \in N_2$ with the minimum cost c_{ij} among all arcs in $A(i)$. As a result, some nodes in N_2 might be unassigned while some other nodes are overassigned (i.e., assigned to more than one node in N_1). The algorithm then gradually converts this solution to a feasible assignment while always maintaining the reduced cost optimality condition. At each iteration the algorithm selects an overassigned node k in N_2, obtains shortest path distances from node k to all other nodes in the residual network with reduced costs as arc lengths, updates node potentials, and augments a unit flow from node k to an unassigned node in N_2 along the shortest path. Since each iteration assigns one more node in N_2 and never converts any assigned node into an unassigned node, within n_1 such iterations, the algorithm obtains a feasible assignment. The relaxation algorithm maintains optimality conditions throughout. Therefore, the shortest path problems have nonnegative arc lengths, and the overall running time of the algorithm is $O(n_1 S(n, m, C))$.

Cost Scaling Algorithm

This algorithm is an adaptation of the cost scaling algorithm for the minimum cost flow problem discussed in Section 10.3. Recall that the cost scaling algorithm performs $O(\log(nC))$ scaling phases and the generic implementation requires $O(n^2 m)$ time for each scaling phase. The bottleneck operation in each scaling phase is performing nonsaturating pushes which require $O(n^2 m)$ time; all other operations, such as finding admissible arcs and performing saturating pushes, require $O(nm)$ time. When we apply the cost scaling algorithm to the assignment problem, each push is a saturating push since each arc capacity is 1. Consequently, the cost scaling algorithm solves the assignment problem in $O(nm \log(nC))$ time.

A modified version of the cost scaling algorithm has an improved running time of $O(\sqrt{n}m \log(nC))$, which is the best available time bound for assignment problems satisfying the similarity assumption. This improvement rests on decomposing the computations in each scaling phase into two subphases. In the first subphase, we apply the usual cost scaling algorithm with the difference that whenever we have relabeled a node more than $2\sqrt{n}$ times, we set this node aside and do not examine it further. When we have set aside all (remaining) active nodes, we initiate the second subphase. It is possible to show that the first subphase requires $O(\sqrt{n_1}m)$ time, and

when it ends, the network will contain at most $O(\sqrt{n_1})$ active nodes. The second subphase makes these active nodes inactive by identifying "approximate shortest paths" from nodes with excesses to nodes with deficits and augmenting unit flow along these paths. The algorithm uses Dial's algorithm (described in Section 4.6) to identify each such path in $O(m)$ time. Consequently, the second subphase also runs in $O(\sqrt{n_1} m \log(nC))$. We provide a reference for this algorithm in the reference notes.

We summarize the preceding discussion.

Theorem 12.2. *The successive shortest path algorithm, Hungarian algorithm, and the relaxation algorithm solve the assignment problem in $O(n_1 S(n, m, C))$ time. A straightforward implementation of the cost scaling algorithm solves the assignment problem in $O(nm \log(nC))$ time and a further improvement of this algorithm runs in $O(\sqrt{n_1} m \log(nC))$ time.* ◆

12.5 STABLE MARRIAGE PROBLEM

The stable marriage problem is a novel application of bipartite matchings. This problem can be stated as follows. A certain community consists of n men and n women. Each person ranks those of the opposite sex in accordance with his or her preferences for a spouse. For a given matching, a man–woman pair is said to be *unstable* if they are not married to each other but prefer each other to their current spouses. A perfect matching (marriage) of men and women is said to be *stable* if it contains no unstable pairs. The stable marriage problem is to identify a stable perfect matching. In this section we show that for *any* set of rankings, we can always find a stable matching. We establish this result constructively, specifying an algorithm that constructs a stable matching in $O(n^2)$ time.

The input to the stable marriage problem consists of two $n \times n$ matrices; the first matrix gives each man's ranking of women and the second matrix gives each woman's ranking of men. A higher rank denotes a more favored person. Without any loss of generality, we can assume that each rank is an integer between 1 and n. To implement the stable marriage algorithm efficiently, we use these two matrices to construct a vector of n elements for each person, called his or her *priority list*, that lists the persons of opposite sex in decreasing order of their rankings. Since all the ranks are between 1 and n, we can construct these priority lists in a total of $O(n^2)$ time using a bucket sort algorithm (see Exercise 12.30).

The algorithm for the stable marriage problem is an iterative greedy algorithm: Each man proposes to his most preferred woman, and each woman receiving more than one proposal rejects all except her most preferred man from among those who have proposed to her. The algorithm maintains a set, LIST, of unassigned men and for each man it maintains an index, called *current-woman*, which denotes the woman in his priority list that he will next offer a proposal. Initially, LIST = N_1, the set of all men, and the *current-woman* of each man is the first woman in his priority list.

The stable marriage algorithm proceeds as follows. At each iteration, the algorithm selects a man from LIST, say Bill, and he proposes to his *current-woman*, say Helen. If Helen is still unassigned, she accepts the proposal and Bill and Helen

are tentatively assigned to each other—they are "engaged." If Helen is already engaged to some man, say Frank, she accepts the proposal of Bill or Frank that she prefers the most and rejects the other. The rejected man designates the next woman on his priority list his *current-woman*. Whenever the algorithm selects a man from LIST, he is removed from it; and whenever a man is rejected by a woman, he is added to LIST. The algorithm repeats this iterative step until LIST is empty, at which point it has assigned all the men and women. We refer to this algorithm as the *propose-and-reject algorithm*.

It is easy to show that the matching obtained by this algorithm is stable. Suppose that Dick prefers Laura to his marriage partner; he must have proposed to Laura at some earlier stage and she must have rejected his proposal in favor of someone whom she liked more than Dick. Consequently, since no woman ever switches to a man that she prefers less, Laura prefers her husband to Dick, so the matching is stable.

To analyze the complexity of the stable marriage algorithm, we note that at each iteration each woman receiving a proposal either (1) receives her first proposal (which occurs exactly once for each woman), or (2) rejects some proposal. Since each woman rejects any man's proposal at most once, the second outcome occurs at most $(n - 1)$ times for each woman. Therefore, the algorithm performs $O(n)$ steps per woman and $O(n^2)$ steps in total. Notice that no algorithm for the stable marriage problem can have any better complexity bound, since the running time of the propose-and-reject algorithm is linear in the length of the input data. We have thus established the following result.

Theorem 12.3. *For any matrix of rankings, the stable marriage problem always has a stable matching. Further, the propose-and-reject algorithm constructs a stable matching in $O(n^2)$ time.* ◆

Needless to say, there could be several stable matchings; the propose-and-reject algorithm constructs one such stable matching. We refer to a pair (i, j) of a man i and a woman j as *stable partners* if some stable matching matches man i with woman j. The matching constructed by our algorithm possesses an interesting property that every man is at least as well off under it as under *any* stable matching. In other words, each man obtains his best possible stable partner. For obvious reasons we refer to such a matching as the *man-optimal* matching. The fact that the matching constructed by our algorithm is a man-optimal matching relies on the following result.

Lemma 12.4. *In the propose-and-reject algorithm, a woman never rejects a stable partner.*

Proof. Let M^* be the matching constructed by the propose-and-reject algorithm. Suppose that the lemma is false and women do reject stable partners. Consider the first time that a woman, say Joan, rejects a stable partner, say Dave. Let M° be the stable matching in which Joan and Dave constitute a stable pair of partners. Suppose that the rejection took place because Joan was engaged to Steve, whom she prefers to Dave. Now notice that prior to the rejection, no other woman had rejected a stable partner, which implies that Steve can have no stable partners whom he prefers to Joan. In M°, let Sue and Steve be the stable pair for Steve. By our

prior observation, Steve prefers Joan to Sue. We have earlier shown that Sue prefers Steve to Dave. The preceding two facts contradict the assumption that M° is a stable matching. This conclusion implies the lemma. ◆

In the propose-and-reject algorithm, men propose to women in decreasing order of their preferences, and since no woman ever rejects a stable partner, each man must be married to the best possible stable partner. Therefore, we have established the following theorem.

Theorem 12.5. *The propose-and-reject algorithm constructs a man-optimal stable matching.* ◆

This theorem is a surprising result. It implies that if each man is independently given his best stable partner, the result is a stable matching. However, we gain this optimality from the men's point of view at the expense of the women. In fact, it is possible to show that in a man-optimal matching, each woman obtains the worst partner that she can have in any stable matching (see Exercise 12.27).

As a concluding remark, we point out that the stable marriage problem also has "nonmatrimonial" applications, such as assigning residents to hospitals, or assigning graduate students to doctoral programs. These applications are actually many-to-one matchings, but can be solved by a minor variation of the propose-and-reject algorithm (see Exercise 12.31).

12.6 NONBIPARTITE CARDINALITY MATCHING PROBLEM

In this section we study the nonbipartite cardinality matching problem on undirected graphs, which we subsequently refer to by the abbreviated name the "nonbipartite matching problem." As we shall see, the nonbipartite matching problem is substantially more difficult to solve than the bipartite problem. To highlight the essential differences between the nonbipartite and bipartite matching problems, we first consider a very natural approach for the matching problem that closely resembles the augmenting path algorithm for solving maximum flow problems discussed in Section 6.4. We show that this approach gives an optimal algorithm for the bipartite problem, but fails for the nonbipartite case. We then identify the reason why the algorithm fails and modify it so that it works for the nonbipartite case as well.

In this section, as usual, we let $A(i)$ denote the node adjacency list of node i [i.e., $A(i) = \{j \in N : (i, j) \in A\}$]. We assume that we store each adjacency list as a singly linked list so that we can insert items into the list in $O(1)$ time. We begin by introducing some notation.

Matched Arcs and Nodes

A *matching* M of a graph $G = (N, A)$ is a subset of arcs with the property that no two arcs of M are incident to the same node. We refer to the arcs in M as *matched arcs*, and arcs not in M as *unmatched arcs*. We also refer to the nodes incident to matched arcs as *matched nodes* and refer to the other nodes as *unmatched*. If (i, j)

belongs to the matching, we say that node i is matched to node j and node j is matched to node i. Figure 12.5 illustrates these definitions. The arcs $\{(2, 4), (3, 5)\}$ constitute a matching in this graph; we depict matched arcs using thicker lines. Notice each node has degree 0 or 1 in the subgraph defined by the matched arcs. Also notice that a matching can contain at most $\lfloor n/2 \rfloor$ arcs.

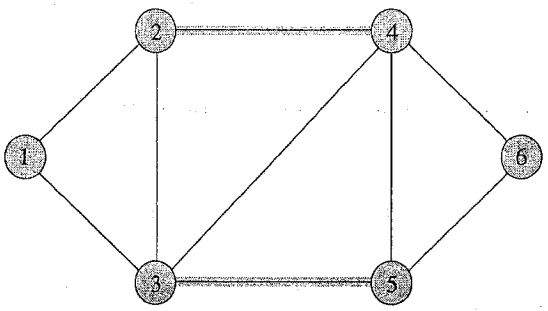


Figure 12.5 Matching example.

Alternating Paths and Cycles

We refer to a path $P = i_1 - i_2 - \cdots - i_k$ in the graph as an *alternating path* with respect to a matching M if every consecutive pair of arcs in the path contains one matched and one unmatched arc. In Figure 12.5, 1–2–4–3–5 and 1–2–4–3–5–6 are alternating paths. We refer to an alternating path as an *even alternating path* if it contains an even number of arcs and an *odd alternating path* if it contains an odd number of arcs. In the preceding example, the first alternating path is even, and the second alternating path is odd. An *alternating cycle* is an alternating path that starts and ends at the same node. In Figure 12.5, 3–2–4–5–3 is an alternating cycle.

Augmenting Paths

We refer to an odd alternating path P with respect to a matching M as an *augmenting path* if the first and last nodes in the path are unmatched. We use the terminology augmenting path because by redesignating matched arcs on the path as unmatched and unmatched arcs as matched, we obtain another matching of cardinality $| M | + 1$. For example, in Figure 12.5 the path 1–2–4–3–5–6 is an augmenting path with respect to a matching of cardinality 2, and if we interchange the matched and unmatched arcs on this path, we obtain the matching of cardinality 3 shown in Figure 12.6.

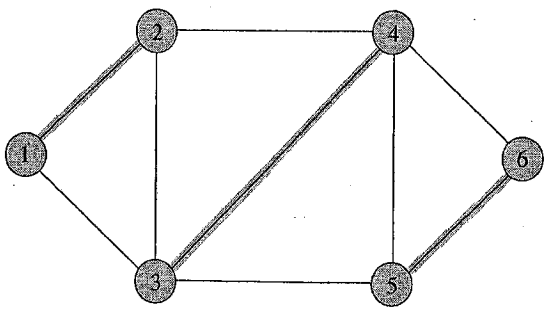


Figure 12.6 Matching a larger cardinality than the matching in Figure 12.5.

Symmetric Difference

The concept of *symmetric difference* of sets is quite important in matching theory. Let S_1 and S_2 be two sets; the symmetric difference of these sets, denoted $S_1 \oplus S_2$, is the set $S_1 \oplus S_2 = (S_1 \cup S_2) - (S_1 \cap S_2)$. In other words, the symmetric difference of sets S_1 and S_2 is the set of elements that are members of one, but not both of S_1 and S_2. For example, if $S_1 = \{4, 5, 7, 8\}$ and $S_2 = \{2, 4, 8, 9\}$, then $S_1 \oplus S_2 = \{2, 5, 7, 9\}$. We shall use the following two properties of symmetric differences.

Property 12.6. *If M is a matching and P is an augmenting path with respect to M, then $M \oplus P$ is a matching of cardinality $|M| + 1$. Moreover, in the matching $M \oplus P$, all the matched nodes in M remain matched and two additional nodes, namely the first and last nodes of P, are matched.* ◆

The symmetric difference of the matching M with the augmenting path P is a set-theoretic way to interchange the matched and unmatched arcs in P, and we have seen earlier that this operation yields a matching of cardinality $M + 1$. We refer to the process of replacing M by $M \oplus P$ as an *augmentation*. The second conclusion of Property 12.6 follows from the definitions.

Property 12.7. *If M and M^* are two matchings, their symmetric difference defines the subgraph $G^* = (N, M \oplus M^*)$ with the property that every component is one of the six types shown in Figure 12.7.* ◆

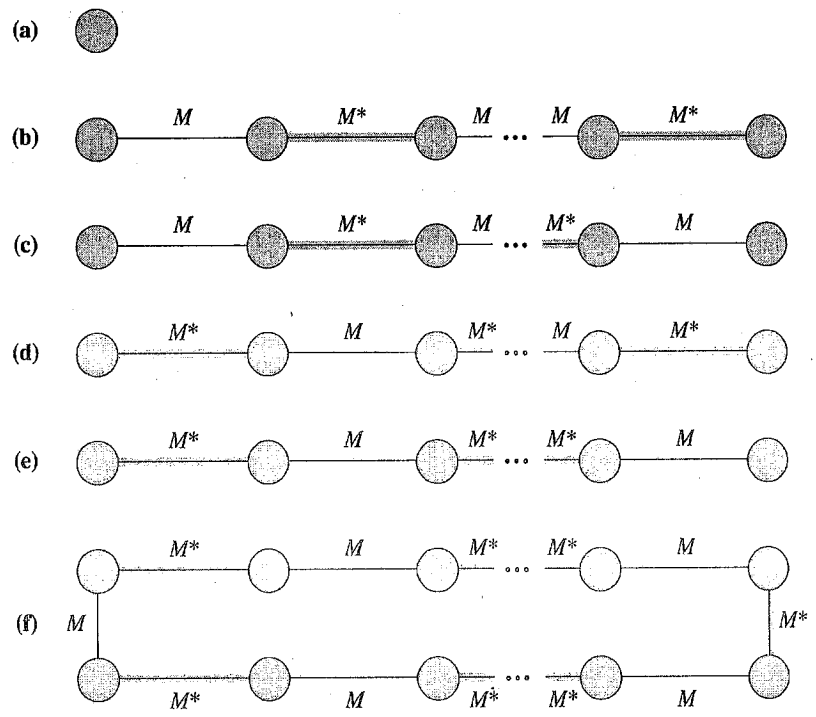


Figure 12.7 Possible types of components formed by a symmetric difference of two matchings M and M^*.

This property follows from the facts that in the subgraph G^* each node has degree 0, 1 or 2, and the only possible components with these node degrees are singleton nodes [as shown in Figure 12.7(a)], paths [as shown in Figure 12.7(b) to (e)], or even-length cycles [as shown in Figure 12.7(f)].

Augmenting Path Theorem

Our algorithm for the matching problem depends crucially on the following augmenting path theorem.

Theorem 12.8 (Augmenting Path Theorem). *If a node p is unmatched in a matching M, and this matching contains no augmenting path that starts at node p, then node p is unmatched in some maximum matching.*

Proof. Let M^* be a maximum matching. If node p is unmatched in M^*, the theorem is clearly true. Therefore, assume that node p is matched in M^*. Consider the symmetric difference of the matchings $M \oplus M^*$. We have seen earlier that every component of the subgraph defined by this symmetric difference is one of the six types shown in Figure 12.7. The fact that node p is unmatched in M rules out all of these possibilities except the ones shown in Figure 12.7(d) and (e) with node p as the starting node. The fact that no augmenting path starts at node p rules out the possibility shown in Figure 12.7(d). Therefore, the only remaining possibility is the even alternating path P shown in Figure 12.7(e) with node p as the starting node. But notice that $M' = M^* \oplus P$ is also a maximum matching in which node p is unmatched. We have thus shown that given a maximum matching M^* in which node p is matched, we can construct another maximum matching M' in which node p is unmatched, which establishes the theorem. ◆

This theorem is an alternative version of a well-known theorem due to Berge, which states that a matching M^* is a maximum matching if and only if the graph G contains no augmenting path with respect to matching M^*. We ask the reader to prove this theorem in Exercise 12.39.

Bipartite Matching Algorithm

The augmenting path theorem suggests the following algorithm for solving the matching problem. Start with a feasible matching M (which might be a null matching) and then repeat the following step for every unmatched node $p \in N$. Try to identify an augmenting path starting at node p. If we find such a path P, replace M with $M \oplus P$; otherwise, delete node p and all the arcs incident to it from the graph.

Using Theorem 12.8, it is easy to show that this algorithm obtains an optimal matching. At each iteration, the algorithm reduces the number of unmatched nodes by at least one, either by deleting a node or by matching it. Since matched nodes remain matched throughout the algorithm (by Property 12.6), when the algorithm terminates, each node in the remaining subgraph, say G', is matched. Consequently, the matching M must be a maximum matching for G'. Theorem 12.8 implies that

the deletion of nodes does not reduce the number of arcs in a maximum cardinality matching. Consequently, M is also a maximum matching in G.

We have therefore reduced the matching algorithm to finding whether or not the network contains an augmenting path starting at node p. How can we find such a path if one exists? The most natural approach might be to use a search algorithm to identify an augmenting path, as we did in the labeling algorithm for the maximum flow problem as discussed in Section 6.5. We can define a node i in the graph as *reachable* from node p if the network contains an alternating path from node p to node i, and then use a search algorithm to identify all reachable nodes. If the algorithm finds an unmatched node that is reachable from node p, it has discovered an augmenting path. However, if none of the reachable nodes is unmatched, we can conclude that the network contains no augmenting path starting at node p.

It is, perhaps, easy to believe that identifying all nodes that are reachable from a specified node should be a rather straightforward task using a search algorithm. Unfortunately, the task is complicated. A straightforward version of a search technique does not work for all matching problems. This approach does work for bipartite matching problems, but fails for nonbipartite problems. Nevertheless, this approach gives valuable insight into the matching problem that will help us in solving the general case. Consequently, we first discuss this straightforward approach and then develop a (nontrivial) modification of it that solves the general problem.

A straightforward approach for solving the matching problem would be to grow a search tree rooted at node p so that each path in the tree from node p to another node is an alternating path. For convenience, we refer to node p as the *root node* of the search tree. For obvious reasons, we also refer to this search tree as an *alternating tree*. We say that the nodes in the alternating tree are *labeled nodes* and that the other nodes are *unlabeled*. The labeled nodes are of two types: *even* or *odd*. Node i is even or odd depending on whether the number of arcs in the unique path from the root node to node i in the alternating tree is even or odd. Notice that whenever an unmatched node (other than the root) has an odd label, the path joining the root node to this node is an augmenting path. For convenience, we assign the label "E" to even nodes and the label "O" to odd nodes.

Recall from Section 3.4 that the search algorithm maintains a set, LIST, of labeled nodes and examines labeled nodes one by one. While examining an even node i, the algorithm scans its adjacency list $A(i)$ and assigns an odd label to every node j in $A(i)$ (provided that node j is unlabeled). If node j is unmatched, we have discovered an augmenting path; otherwise, we add this node to LIST. On the other hand, while examining an odd node i, the algorithm examines its unique matched arc (i, j). If node j is unlabeled, the algorithm assigns an even label to this node and adds it to LIST. The search algorithm terminates when LIST becomes empty, or it has assigned an odd label to an unmatched node, thus discovering an augmenting path.

Figure 12.8 illustrates the process of growing the search tree on the graph shown in Figure 12.8(a). Assuming that we examine the labeled nodes in the first-in, first-out order, and scan the nodes in any adjacency list in increasing order of the node numbers, the algorithm will examine the nodes in the order 1–2–4–3–7–6–8–5. The resulting alternating tree shown in Figure 12.8(b) has an augmenting path 1–4–7–8.

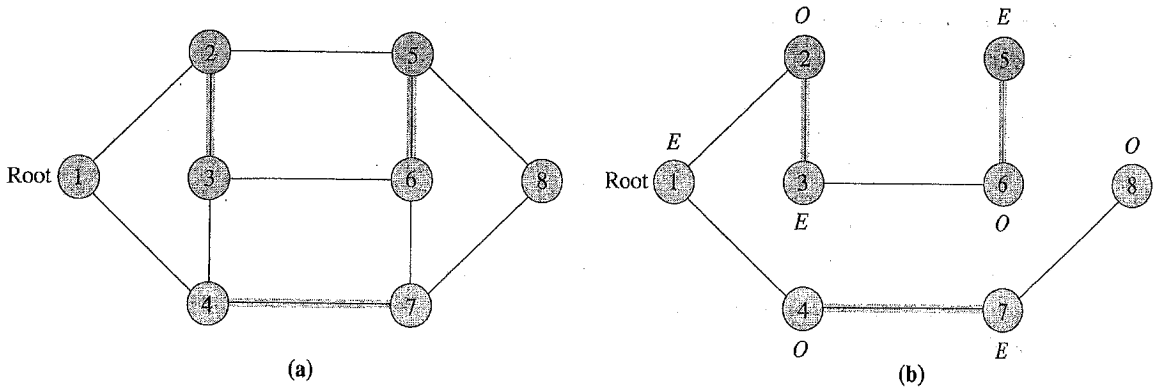


Figure 12.8 Growing an alternating tree: (a) the graph; (b) its complete alternating tree.

We are now in a position to give a complete algorithmic description of the matching algorithm. We subsequently refer to this algorithm, as described in Figures 12.9 and 12.10, as the *bipartite matching algorithm* because, as we explain later, it will always establish a maximum matching in bipartite networks (it might fail when applied to nonbipartite networks).

```
algorithm matching;
begin
    M := ∅;
    for each node p ∈ N do
      if node p is unmatched then
      begin
          search(p, found);
          if found equals true then augment
          else delete node p and all arcs incident to it from G;
      end;
    M* = M;
end;
```

Figure 12.9 Bipartite matching algorithm.

It is easy to show that the matching algorithm runs in $O(nm)$ time. The algorithm executes the search and augment procedures at most n times. The augment procedure clearly requires $O(n)$ time. It is easy to see that the search procedure requires $O(m)$ time per execution. For each node i, the search procedure performs one of the following two operations at most once: (1) it executes examine-even(i, found), or (2) it executes examine-odd(i, found). The latter operation requires $O(1)$ time per execution. The former operation requires $O(|A(i)|)$ time, so a total of $O(\sum_{i \in N} |A(i)|) = O(m)$ time for all the nodes.

Difficulties with the Bipartite Matching Algorithm

Does the search procedure work correctly? It is clear that whenever the algorithm finds an augmenting path starting at node p, this path is an augmenting path. But when the algorithm fails to find an augmenting path, can we conclude that the network contains no such path? We shall show that if the graph possesses a *unique label property* (defined next), our conclusion will be correct; otherwise, the conclusion could be incorrect.

```
procedure search(p, found);
begin
    found : = false;
    unlabel all nodes;
    give an even label to node p and initialize LIST = {p};
    while LIST ≠ ∅ do
    begin
        delete a node i from LIST;
        if node i has an even label then examine-even(i, found)
        else examine-odd(i, found);
        if found equals true then return;
    end;
end;
```

(a)

```
procedure examine-even(i, found);
begin
    for every node j ∈ A(i) do
    begin
        if node j is unmatched then set q : = j and pred(q) : = i;
        found : = true and return;
        if node j is matched and unlabeled then
            set pred(j) : = i, give node j an odd label and add node j to LIST;
    end;
end;
```

(b)

```
procedure examine-odd(i,found);
begin
    let j be the node matched to node i;
    if node j is unlabeled then set pred(j) = i, give node j an even label and add it to LIST;
end;
```

(c)

```
procedure augment;
begin
    trace the augmenting path P by starting at node q and traversing the predecessor indices;
    update the matching using the operation M : = M ⊕ P;
end;
```

(d)

Figure 12.10 Procedures for the bipartite matching algorithm.

Unique label property. *A graph is said to possess a unique label property with respect to a given matching M and a root node p if the search procedure assigns a unique label to every labeled node (i.e., even or odd) irrespective of the order in which it examines labeled nodes.*

It is easy to show that if the graph possesses the unique label property, it will

always discover an augmenting path if one such path exists. Suppose that the network does contain an augmenting path $p - i_1 - j_1 - i_2 - j_2 - \cdots i_l - j_l - q$ from node p to node q with respect to the matching M. If we examine the nodes p, i_1, j_1, i_2, j_2, . . . in order, we will assign even labels to nodes $p, j_1, j_2, \ldots, j_l$, and odd labels to nodes $i_1, i_2, \ldots, i_l, q$. Since the graph possesses the unique label property, the algorithm would assign the same labels no matter in which order the search procedure examines the labeled nodes. Therefore, the search procedure will always assign an odd label to node q and will discover an augmenting path.

Does any network satisfy the unique label property with respect to any matching and any root node? Yes; in fact, bipartite networks satisfy this property. Recall from Section 2.2 that in a bipartite network $G = (N, A)$, we can partition the node set N into two subsets N_1 and N_2 so that every arc $(i, j) \in A$ has its end points in different subsets. For a bipartite network, if the root node is in N_1, every labeled node in N_1 will receive an even label and every labeled node in N_2 will receive an odd label (because the alternating path will begin at a node in N_1 and then alternate between nodes in N_2 and N_1 respectively). Similarly, if the root node is in N_2, every labeled node in N_1 will receive an odd label and every labeled node in N_2 will receive an even label. Consequently, the matching algorithm will find an optimal matching in bipartite networks.

Nonbipartite networks might not satisfy the unique label property, and therefore the search algorithm might fail to detect an augmenting path even though the network contains one. Consider, for example, the situation shown in Figure 12.11. If node 5 receives its label via the path 1–2–3–4–5, it receives the even label. When we examine node 5, the search algorithm gives node 6 an odd label and discovers the augmenting path 1–2–3–4–5–6. However, if node 5 receives its label via the path 1–2–3–7–8–5, its label will be odd. Since node 5 has an odd label, we scan its unique matched arc (5, 4), attempting to label node 4, but do not scan arc (5, 6). Thus the algorithm fails to discover an augmenting path.

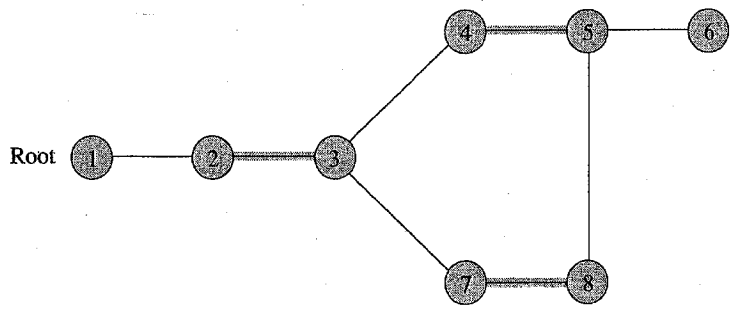


Figure 12.11 Nonbipartite matching problem.

The preceding situation arises because we can connect node 5 to the root by both an odd-length and an even-length alternating path. Therefore, depending on the order in which the search algorithm examines labeled nodes, node 5 might receive an even or an odd label. But since we assign only one label to any node (either even or odd), assigning an odd label prevents us from giving the node an even label in subsequent stages, so we miss the opportunity to give node 6 an odd label.

One plausible way to overcome this difficulty would be to permit nodes to have both even and odd labels. When a node i receives an even label, we scan its adjacency list $A(i)$ to label further nodes; and when a node i receives an odd label, we scan its unique matched arc. But even this modification does not work. To see this, consider the example shown in Figure 12.12. We might examine the nodes in the following order: 1 (even), 2 (odd), 3 (even), 4 (odd), 5 (even), 8 (odd), 7 (even), 3 (odd), 2 (even), 6 (odd). At this point, the unmatched node 6 receives an odd label and the algorithm would declare that it has found an augmenting path, even though the network contains no such path. To summarize, we find that by assigning just one label to each node, we might overlook an augmenting path, and by assigning two labels, we might falsely believe that we have found an augmenting path.

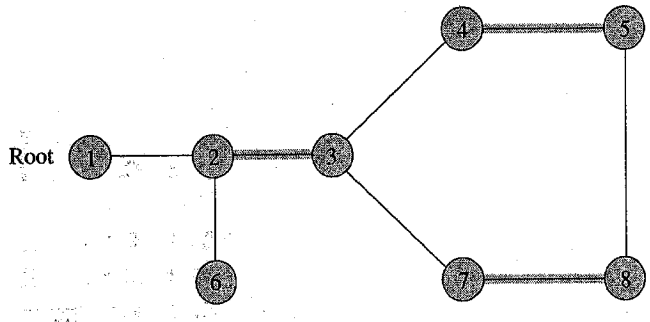


Figure 12.12 Another nonbipartite matching problem.

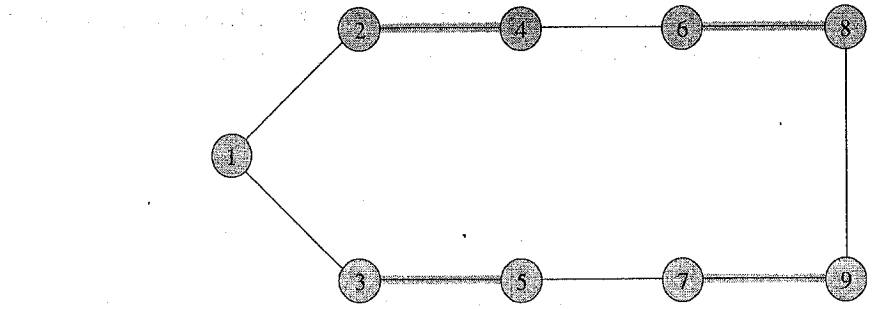
Why do we encounter this difficulty? What causes the algorithm to break down? The root cause of the difficulty in solving a nonbipartite matching problem is the presence of certain subgraphs called *flowers*, composed of particular types of paths and odd cycles. (Note that since bipartite graphs contain no odd cycles, they never contain any flowers.)

Flowers and Blossoms

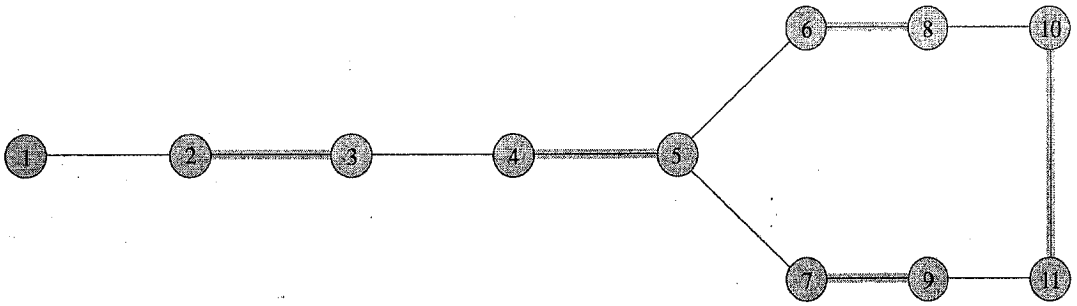
A *flower*, defined with respect to a matching M and a root node p, is a subgraph with two components:

1. *Stem.* A stem is an even (length) alternating path that starts at the root node p and terminates at some node w. We permit the possibility that $p = w$, in which case we say that the stem is empty.
2. *Blossom.* A blossom is an odd (length) alternating cycle that starts and terminates at the terminal node w of a stem and has no other node in common with the stem. We refer to node w as the *base* of the blossom.

Figure 12.13 shows two examples of flowers. The flower shown in Figure 12.13(a) has an empty stem, and the flower shown in Figure 12.13(b) has a nonempty stem. We denote a blossom by B and define it by its set of arcs or set of nodes, whichever is convenient. In our subsequent discussion, we use several properties of flowers, which we record for easy future reference.



(a)



(b)

Figure 12.13 Two examples of flowers.

Property 12.9

(a) A stem spans 2l nodes and contains l matched arcs for some integer $l \geq 0$.
(b) A blossom spans $2k + 1$ *nodes and contains k matched arcs for some integer* $k \geq 1$. *The matched arcs match all nodes of the blossom except its base.*
(c) The base of a blossom is an even node.

Property 12.10. *Every node i in the blossom (except its base) is reachable from the root (or from the base of the blossom) through two distinct alternating paths; one has even length and the other has odd length (see, e.g., Figure 12.14).*

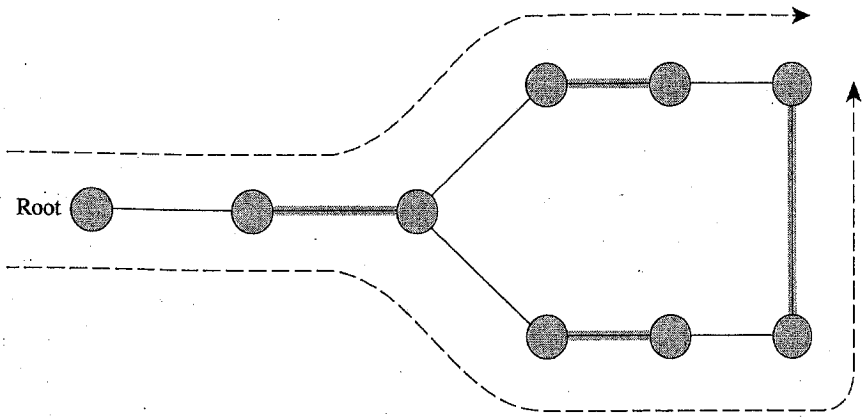


Figure 12.14 Two distinct alternating paths from the root to every node in the blossom.

The even alternating path to node i terminates with a matched arc, and the odd alternating path to node i terminates with an unmatched arc.

These properties are relatively straightforward to establish, so we omit their proofs.

Contracting a Blossom

We now consider the issue we were discussing prior to our definition of blossoms: Why might the labeling algorithm fail to identify an augmenting path, and how might we remedy the problem? If the network contains a blossom with respect to the current matching and the root node p, each node in the blossom is qualified to receive an even label because the network contains an even alternating path from the root to that node. But the search algorithm will give even labels to some nodes in the blossom and odd labels to others. Notice that if we had a choice we would prefer to give even labels to the nodes for the following reason: When examining even-labeled nodes, we can label nodes outside the blossom by searching along all unmatched arcs incident to nodes in the blossom; when examining odd-labeled nodes, however, we label only the nodes in the blossom.

So, it seems intuitively clear that if we could give all the nodes in the blossom an even label, whenever we detect a blossom, the search algorithm would always detect an augmenting path. There are several ways to achieve this objective; one of the more popular approaches is to contract (or shrink) the blossom into a single node. This operation replaces the blossom B consisting of the node sequence $i_1 - i_2 - \cdots - i_k - i_1$ by a single new node b in the following manner:

1. Introduce a new node b and define its adjacency list $A(b) = A(i_1) \cup A(i_2) \cup \cdots \cup A(i_k)$.
2. Update the adjacency list of every node $j \in A(b)$ by executing $A(j) = A(j) \cup \{b\}$.
3. To be able to recover information about the nodes within the blossom that we have contracted into the single node b, we form a circular doubly linked list of nodes $i_1, i_2, \ldots, i_k$, and then delete the nodes $i_1, i_2, \ldots, i_k$ and all arcs incident to these nodes from the network. (Notice that this operation requires the updating of the adjacency list of all the nodes that are adjacent to the deleted nodes.)

We refer to the resulting network $G^c = (N^c, A^c)$ as the *contracted network*. We let $A^c(i)$ denote the adjacency list of a node i in G^c and let M^c denote the corresponding matching in the contracted network.

Figure 12.15(a) illustrates a contraction. The flower 1–2–3–4–5–6–7–3 in this figure contains the blossom 3–4–5–6–7–3. Contracting all these nodes into a new node, node 11, we obtain the graph shown in Figure 12.15(b). Each contraction operation creates a new node. To differentiate this node from the nodes of the original network, we refer to it as a *pseudonode*. Notice that a pseudonode is always an even node because it merges the entire blossom into its base, which is always even [see Property 12.9(c)]. Since the adjacency list of the pseudonode is the union of

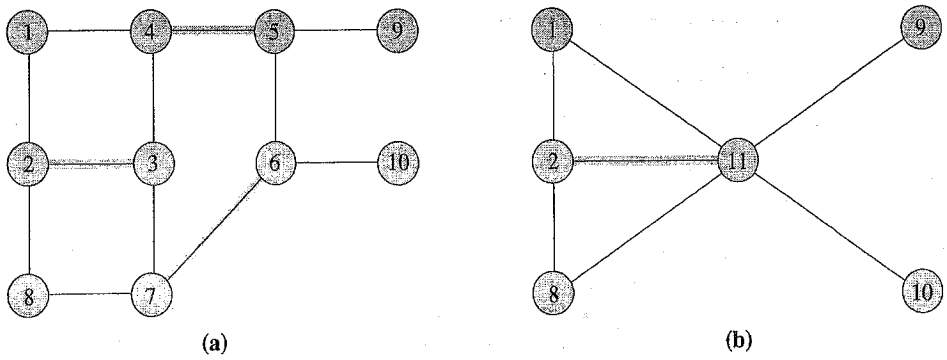


Figure 12.15 Contraction: (a) network before the contraction; (b) network after the contraction.

the adjacency lists of the nodes it contains, scanning the adjacency list of the pseudonode allows us to reach out to all the nodes that we would have reached out to from nodes in the blossom. Consequently, contracting the entire blossom into a single even pseudonode amounts to assigning even labels to each blossom node in the original graph.

Nonbipartite Matching Algorithm

We are now in a position to describe the nonbipartite matching algorithm. This algorithm modifies the search procedure of the bipartite matching algorithm in the following manner. As the search procedure proceeds, it assigns even or odd labels to the nodes. Although the algorithm will never relabel an already labeled node, it will identify the possibility of assigning an odd label to a node with an even label, or of assigning an even label to a node with an odd label. When we find that we can, for the first time, assign a node, say node i, a label other than what it already has, we suspend the search procedure. At this point we have discovered an even as well as an odd alternating path to node i. If we trace back the predecessor indices of these paths until we encounter the first common node on these paths, the arcs we have traced constitute a blossom and the first common node (which has an even label) is the base of the blossom (see Exercise 12.37). We then contract the blossom into a pseudonode, update the data structures, and continue the search procedure. We might note that we could perform several contractions before we either discover an augmenting path (in the contracted graph) or run out of nodes to examine, which indicates that the network contains no augmenting path starting from the root node p. If we succeed in identifying an augmenting path from node p to some unmatched node q, we check whether this path contains any pseudonodes. If so, we expand the blossoms represented by these pseudonodes one by one, in an order to be described later, until the augmenting path contains no pseudonodes. We point out that we can contract blossoms containing pseudonodes; so pseudonodes might contain other pseudonodes.

The algorithmic description of the resulting algorithm, which we subsequently refer to as the *nonbipartite matching algorithm*, is the same as the bipartite matching algorithm with the exception of a change in the search procedure. Figure 12.16 shows

```
procedure search(p, found);
begin
    set A^c(i) : = A(i) for all nodes i;
    found : = false;
    unlabel all nodes;
    give an even label to node p and initialize LIST : = {p};
    while LIST ≠ ∅ do
    begin
        delete a node i from LIST;
        if node i has even label then examine-even(i, found)
        else examine-odd(i, found);
        if found = true then return;
    end;
end;
```

(a)

```
procedure examine-even(i, found);
begin
    for every node j ∈ A^c(i) do
    begin
        if node j has an even label then contract(i, j) and return;
        if node j is unmatched then set q : = j,
            pred(q) : = i, found : = true and return;
        if node j is matched and unlabeled then set pred(j) : = i,
            give node j an odd label and add it to LIST;
    end;
end;
```

(b)

```
procedure examine-odd(i, found);
begin
    let node i be matched to node j;
    if node j has an odd label then contract(i, j) and return;
    if node j is unmatched and unlabeled
        then set pred(j) : = i, give node j an even label and add it to LIST;
end;
```

(c)

```
procedure contract(i, j);
begin
    trace the predecessor indices of nodes i and j to identify a blossom B;
    create a new node b and define A^c(b) = ∪_{k∈B} A^c(k);
    give an even label to node b and add it to LIST;
    update A^c(j) = A^c(j) ∪ {b} for each j ∈ A^c(b);
    form a circular doubly linked list of nodes in B;
    delete the nodes in B from the network and update the data structure;
end;
```

(d)

Figure 12.16 Procedures for the nonbipartite matching algorithm.

```
procedure augment;
begin
    trace the augmenting path P' by starting at node q and
      traversing the predecessor indices;
    if the path P' contains pseudonodes then expand the corresponding
      blossoms and obtain an augmenting path P in the original network;
    update the matching using the operation M = M ⊕ P;
end;
```

(e)

Figure 12.16 (*Continued*)

the new search procedure and its subroutines. Notice that the algorithm uses an additional procedure *contract* that contracts a blossom into a pseudonode.

In the nonbipartite matching algorithm, each time we execute the procedure *contract*, we create a new pseudonode. One particularly simple scheme for keeping track of these additional nodes would be to number them as $n + 1$, $n + 2$, $n + 3$, ... In this scheme, a node i is a pseudonode if and only if $i > n$.

We illustrate the nonbipartite matching algorithm by applying it to the numerical example shown in Figure 12.17(a). We assume that the algorithm examines the labeled nodes in the first-in, first-out order, and scans the adjacency list of any node in increasing order of the node numbers. Suppose that the algorithm selects node 1

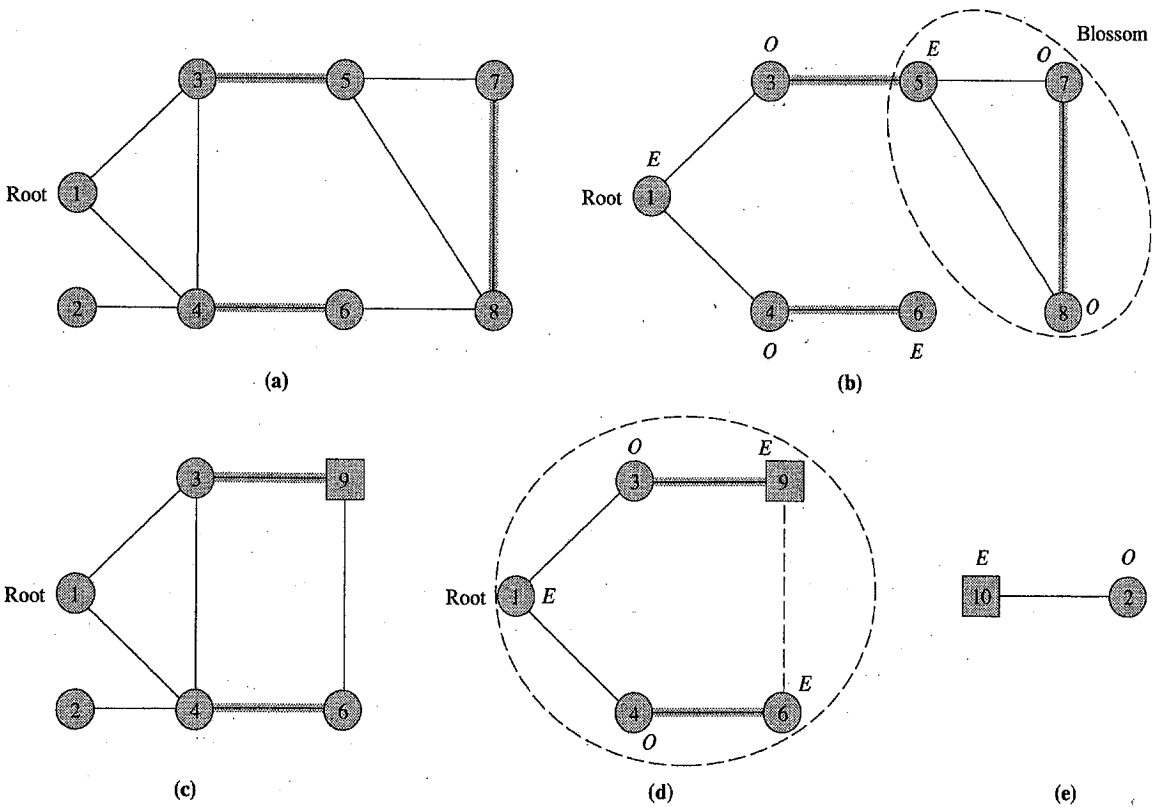


Figure 12.17 Identifying an augmenting path in the contracted network: (a) example network; (b) alternating tree; (c) contracted graph; (d) blossom; (e) contracted graph.

as the root node. Then it would examine the nodes in the following order: 1 (even), 3 (odd), 4 (odd), 5 (even), 6 (even), 7 (odd). Figure 12.17(b) shows the alternating tree at this point. While examining node 7, the algorithm scans arc (7, 8) and discovers the blossom 5–7–8–5. Contracting this blossom into the pseudonode numbered 9 gives us the contracted graph shown in Figure 12.17(c). To distinguish a pseudonode from a node of the original network, in the figure, we depict a pseudonode as a square instead of a circle. At this point, node 9 is the only unexamined node; while examining the adjacency list of node 9, we discover another blossom spanning the nodes 1–3–9–6–4–1 [see Figure 12.17(d)]. Contracting this blossom into the pseudonode numbered 10 gives us the contracted graph shown in Figure 12.17(e). While examining node 10, we assign an odd label to the unmatched node 2 and discover an augmenting path 10–2.

We now expand the pseudonodes in the augmenting path so that we can find an augmenting path in the original network. We first expand node 10, as shown in Figure 12.18(b). Node 2 is adjacent to the blossom node 4. To the arc (2, 4), we add the even alternating path from the root to node 4. Doing so gives us the path 1–3–9–6–4–2. We next expand node 9, as shown in Figure 12.18(c), and obtain the augmenting path 1–3–5–7–8–6–4–2 in the original network.

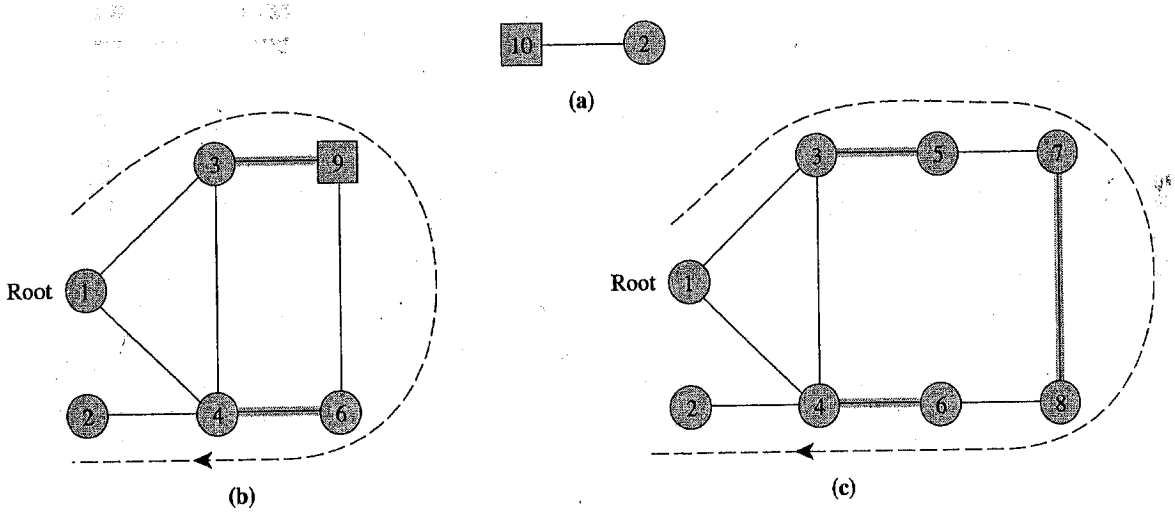


Figure 12.18 Identifying an augmenting path in the original network: (a) contracted graph; (b) network after expanding node 10; (c) network after expanding node 9.

Correctness of the Nonbipartite Matching Algorithm

To show that the algorithm correctly finds a maximum matching, we need to show that (1) whenever we find an augmenting path in the contracted graph we can, by expanding the contracted nodes, also find an augmenting path in the original network; and (2) by contracting blossoms we do not add or omit augmenting paths. To prove this result, we assume that we contract only one blossom. If we do contract more than one blossom, we can use this result iteratively to prove the validity of multiple contractions. In the proof of the theorem, we assume that we have contracted a blossom B with respect to a matching M whose base is w, creating the pseudonode

b. We let G^c and M^c respectively represent the contracted graph and the matching in the contracted graph.

Lemma 12.11. *If the contracted network G^c contains an augmenting path P^c starting at the root node p (or the pseudonode containing p) with respect to the matching M^c, then the original network G contains an augmenting path starting at the root p with respect to the matching M.*

Proof. If the augmenting path P^c does not contain node b, it also is an augmenting path in G and the conclusion is valid. Next suppose that b is an interior node of P^c (i.e., the blossom B has a nonempty stem). In that case the augmenting path in the contracted network will have the structure shown in Figure 12.19(a). Recall from Property 12.9(c) that the pseudonode b is an even node and the alternating path from node p to b ends with a matched arc. We can represent the augmenting path in G^c as $[P_1, (i, b), (b, l), P_3]$. If we expand the contracted node, we obtain the graph shown in Figure 12.19(b). Notice that node l is incident to some node in the blossom, say node k. Property 12.10 implies that the network contains an even alternating path from node w (i.e., the base of the blossom) to node k that ends with a matched arc. Let P_2 denote this path. Now observe that the path $[P_1, (i, w), P_2, (k, l), P_3]$ is an augmenting path in the graph G. This result establishes the lemma whenever b is an internal node of the augmenting path. Whenever b is the first node of the augmenting path, $p = w$ and the path $[P_2, (k, l), P_3]$ is an augmenting path in the original graph. ◆

This lemma shows that if we discover an augmenting path in the contracted network, we can use this path to identify an augmenting path in the original network. The lemma also shows that by contracting a blossom we do not add any augmenting paths beyond those that are contained in the original graph. We now need to prove the converse result: If G contains an augmenting path in G from node p to some node q with respect to the matching M, then G^c also contains an augmenting path from node p (or the pseudonode containing p) to node q with respect to the matching M^c. This result will show that by contracting nodes, we do not miss any augmenting paths from the original network.

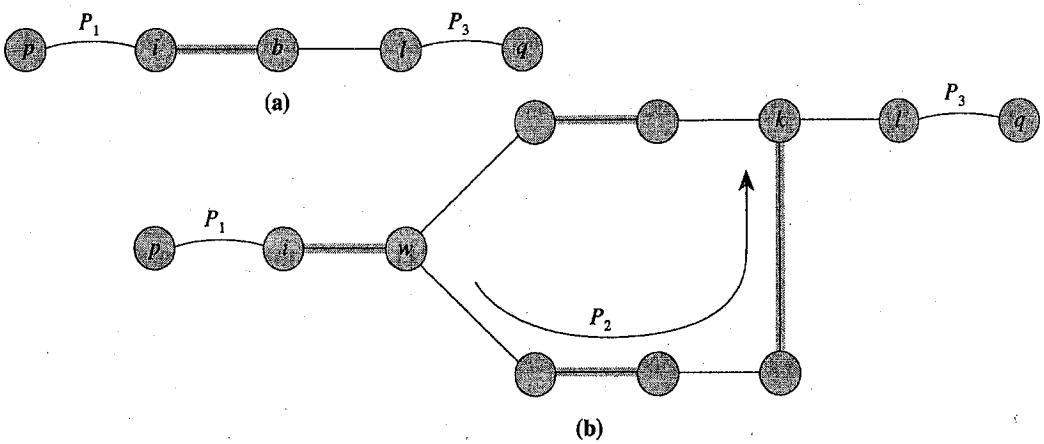


Figure 12.19 Identifying an augmenting path in the original network: (a) augmenting path in contracted network; (b) augmenting path in original network.

Lemma 12.12. *If G contains an augmenting path from node p to node q with respect to a matching M, then G^c contains an augmenting path from node p (or the pseudonode containing p) to node q with respect to the matching M^c.*

Proof. Suppose that G contains an augmenting path P from node p to node q with respect to a matching M and that nodes p and q are the only unmatched nodes in G. We incur no loss of generality in making this assumption since nodes p and q are the only unmatched nodes that appear in P, so this path remains an augmenting path even if we delete the remaining unmatched nodes. If the path P has no node in common with the nodes in the blossom B, we have nothing to prove because P is also an augmenting path in the contracted network. When P has some nodes in common with the blossom B, we consider two cases:

Case 1: The blossom B has an empty stem. In this case, node p is the base of the blossom and the pseudonode b in the contracted network contains node p. Let node i be the last node of the path P that lies in the blossom. Path P has the form $[P_1, (i, j), P_2]$ for some node j and some unmatched arc (i, j) [see Figure 12.20(a)]. Note that the path P_1 might have some arcs in common with the blossom. Now notice that $[(b, j), P_2]$ is an augmenting path in the contracted network and we have established the desired conclusion [see Figure 12.20(b)].

Case 2: The blossom B has a nonempty stem. Let P_3 denote the even alternating path from node p to the base w of the blossom and consider the matching $M' = M \oplus P_3$. In the matching M', node p is matched and node w is unmatched. Moreover, since the matchings M and M' have the same cardinality, M is not a maximum matching if and only if M' is not a maximum matching. By assumption, G contains an augmenting path with respect to M. Therefore, G must also contain an augmenting path with respect to M'. But with respect to the matching M', nodes w and q are the only unmatched nodes in G, so the network must contain an augmenting path between these two nodes.

Now let $M^{c'}$ denote the matching in the contracted graph G^c corresponding to the matching M' in the graph G. Note that M^c might be different than $M^{c'}$. In the

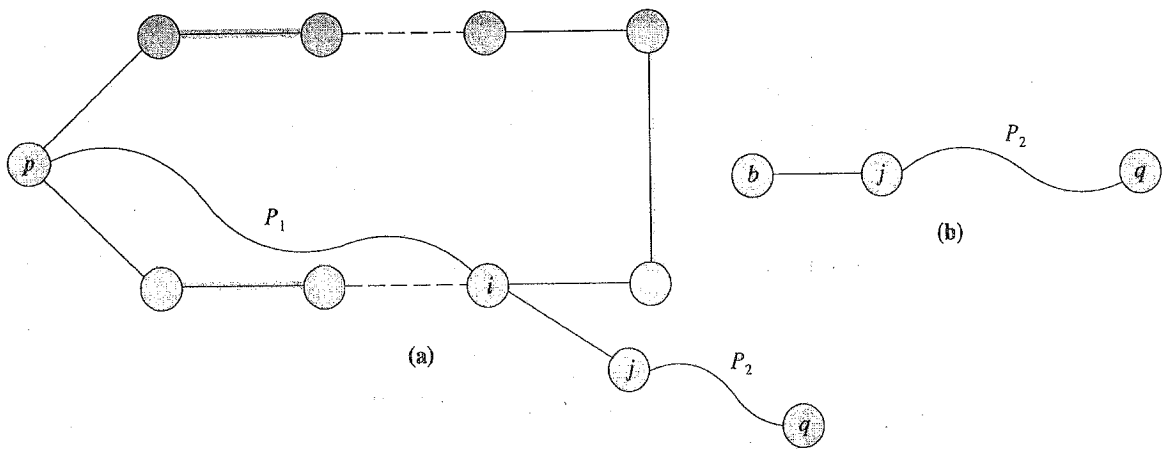


Figure 12.20 Proving case 1 of Lemma 12.12: (a) augmenting path in original network; (b) augmenting path in contracted network.

matching M', the blossom B has an empty stem and the analysis of Case 1 implies that the graph contains an augmenting path after we contract the nodes of the blossom. Consequently, G^c contains an augmenting path with respect to the matching $M^{c'}$. But since $M^{c'}$ and M^c have the same cardinality, G^c must also contain an augmenting path with respect to the matching M^c. This conclusion completes the proof of the lemma. ◆

The preceding two lemmas show that the contracted network contains an augmenting path starting at node p if and only if the original network contains one. Therefore, by performing contractions we do not create new augmenting paths containing node p nor do we miss any. As a consequence, the nonbipartite matching algorithm correctly computes a maximum matching in the network.

Complexity of the Nonbipartite Matching Algorithm

We next show that the nonbipartite matching algorithm has a worst-case complexity of $O(n^3)$. As a first step in establishing this result, we obtain a bound on the total number of contractions we can conduct in one execution of the search procedure.

Lemma 12.13. *During an execution of the search procedure, the algorithm performs at most $n/2$ contractions.*

Proof. A blossom contains at least three nodes, and contracting it produces one new node. So each contraction reduces the number of nodes by at least 2. Since the network initially contains n nodes, we can perform at most $n/2$ contractions between two augmentations. ◆

The nonbipartite matching algorithm is the same as the bipartite matching algorithm except that it contracts and expands blossoms while executing the search procedure. To perform these steps efficiently, we need to contract blossoms cleverly so that we can perform future expansions easily. Achieving this objective requires that we slightly modify our earlier method for contracting a blossom. The modified method for contracting blossoms is exactly the same as the earlier method except that we do not delete the nodes $i_1, i_2, \ldots, i_k$ of the blossom B from the network since this operation would require that we make many changes to our data structures. Instead, we formally keep these nodes as part of the network, but declare them as *inactive*, so that we avoid examining them in future steps. The advantage of this modification is that it contracts the network while maintaining information about the original network that we use later to expand the blossoms. In our subsequent discussion we refer to those nodes that are not inactive as *active* nodes. We also refer to an arc (i, j) as an *active arc* if nodes i and j are both active; otherwise, we refer to it as an *inactive arc*.

By not deleting the nodes that we have contracted into pseudonodes, we need to exercise some care in carrying out and analyzing the algorithm. First, the algorithm might attempt to examine inactive nodes. This possibility poses no problem, however, since we can check the status of a node or arc before examining it and ignore the node or arc if it is inactive. As a second consideration, we note that keeping

inactive nodes in the network increases the size of the adjacency list of some nodes, since whenever we contract a set of nodes into a pseudonode, all the nodes that are adjacent to the contracted nodes will now also have the pseudonode as a neighbor. This increase in the size of the adjacency lists might increase the execution time of certain steps. Since each contraction adds at most one element to any adjacency list (the pseudonode), and since the algorithm performs at most $n/2$ contractions, no adjacency list will ever contain more than $3n/2$ elements. Therefore, the increase in size of the adjacency lists will not add to the computational complexity of the algorithm.

We have now given sufficient background material for carrying out the worst-case analysis of the nonbipartite matching algorithm. We intend to show that each execution of the search and augment procedure requires $O(n^2)$ time. Since the nonbipartite matching algorithm executes these procedures at most n times, the overall algorithm runs in $O(n^3)$ time.

First, consider the time that the algorithm spends without contracting and expanding blossoms. For each node i, the search procedure performs one of the following operations at most once: (1) it discovers that node i is inactive, in which case it does nothing; (2) it executes examine-odd(i, found), or (3) it executes examine-even(i, found). Clearly, the first two cases require $O(1)$ time; the time for the third step, however, is proportional to $| A^c(i) | \leq 3n/2$. Since the search procedure examines at most $3n/2$ nodes, we obtain a bound of $O(n^2)$ on its running time, ignoring the time for handling blossoms.

Our next task is to analyze the time for contracting blossoms. The bottleneck step in contracting a blossom B consisting of the node sequence $i_1 - i_2 - \cdots - i_k - i_1$ is to form the adjacency list of the resulting pseudonode b, which is defined as $A^c(b) = A^c(i_1) \cup A^c(i_2) \cup \cdots \cup A^c(i_k)$. To construct the adjacency list $A^c(b)$, we first use a "marking" method for finding the nodes that will be adjacent to the pseudonode. We first declare all nodes in the network as unmarked. Then we examine nodes in the blossom one by one, and for each node i being examined, we mark all the nodes in $A^c(i)$. When we have examined all the nodes in the blossom, we again scan all the nodes in the network and form a set of marked nodes. These are exactly the nodes that are adjacent to the pseudonode b. We record these nodes as $A^c(b)$ and we also add node b to the adjacency list of all these nodes. Clearly, this method requires $O(n)$ effort each time we contract a blossom (we do so at most $n/2$ times) plus the time spent in scanning nodes in the adjacency list $A^c(i)$. The latter time also sums to $O(n^2)$ over all contractions, because each node is part of a blossom at most once (since it becomes inactive subsequently), so the algorithm will examine its adjacency list at most once.

Finally, we analyze the time required for expanding blossoms. The search procedure needs to expand blossoms during an augmentation in order to obtain an augmenting path in the original network. Suppose that the procedure discovers an augmenting path P from node p to node q in the contracted network. We determine a corresponding augmenting path in the original network using the following repetitive process: we start at node q and trace back the predecessor indices until either (1) we arrive at node p, or (2) we encounter a pseudonode; in the latter case, we expand the blossom and obtain a corresponding augmenting path in the expanded network. After at most $n/2$ such iterations, we obtain an augmenting path in the

original network. Let us show that we can expand each pseudonode in $O(n)$ time, which would establish a time bound of $O(n^2)$ per augmentation.

Suppose that node j is the first pseudonode encountered in the path while tracing predecessor indices from node q. Let $\text{pred}(i) = j$. By definition, node i is not a pseudonode. Clearly, node i is adjacent to some node in the blossom B contained in the pseudonode j, and this node must be contained in the adjacency list $A^c(i)$. To locate this node, we again use a marking approach; we first declare all the nodes in the network as unmarked, mark all nodes in the blossom B, and then scan the nodes in $A^c(i)$ to identify a marked node. Let node k be such a node. Node k is incident to two arcs in the blossom; one of these arcs is matched and the other is unmatched. We then trace through the nodes of the blossom in the direction of the matched arc until we reach the base of the blossom, at which point we trace the predecessor indices to reach node p. The result is an augmenting path from node p to node q in the expanded network; as is clear from the preceding discussion, this method requires $O(n)$ time. We might note that if during the course of expanding a blossom, we encounter a pseudonode, we expand this pseudonode by the method we have just described, and once we have finished expanding this node, we continue to expand the pseudonode that contained it.

We have now shown that all the steps of the search procedure require $O(n^2)$ time per execution. The matching algorithm calls the search procedure at most n times and therefore runs in $O(n^3)$ time. We state this result as a theorem.

Theorem 12.14. *The bipartite matching algorithm identifies a maximum matching in a network in $O(n^3)$ time.* ◆

12.7 MATCHINGS AND PATHS

In this section we describe some interesting relationships between matchings and paths. We show how to solve the shortest path problem in a directed network between a specific pair of nodes by solving two assignment problems. In fact, this transformation allows us to obtain the best current time bound for solving the shortest path problem with arbitrary arc lengths. We also show how to solve a shortest path problem in an undirected network with arbitrary arc lengths by solving a nonbipartite weighted matching problem.

Shortest Paths in Directed Networks

Suppose that we want to determine a shortest path from node s to node t in a directed network that might contain negative arc lengths. We will solve this problem by invoking two applications of any algorithm for the assignment problem. The first application determines if the network contains a negative cycle; if it does not, the second application identifies a shortest path. To solve the assignment problem, we can use $O(n^{1/2}m \log(nC))$ time approach that we outlined in Section 12.4.

Consider a shortest path problem in the network $G = (N, A)$. We apply the node-splitting transformation on this network and replace each node i by two nodes i and i'. Furthermore, we replace each arc (i, j) by an arc (i, j') and add an *artificial* zero cost arc (i, i'). As an illustration, consider the shortest path problem from node

1 to node 5 shown in Figure 12.21(a). Figure 12.21(b) gives the transformed network of Figure 12.21(a). We first note that the transformed network always has a feasible assignment with cost zero, namely, the assignment containing all artificial arcs. We next show that the optimal value of the assignment problem in the transformed network is negative if and only if the original network has a negative cycle.

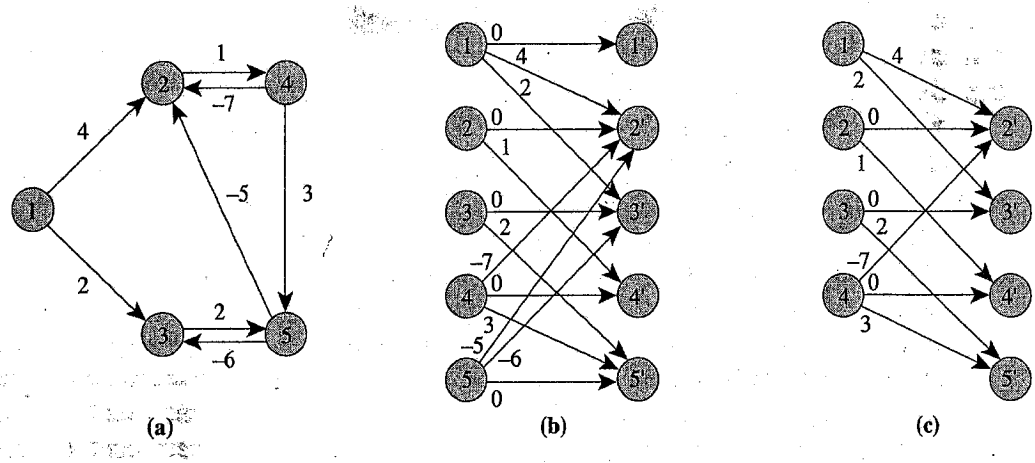


Figure 12.21 Transforming a shortest path problem to an assignment problem: (a) original network; (b) network for identifying a negative cycle; (c) network for identifying a shortest path from node 1 to node 6.

First, suppose that the original network contains a negative cost cycle $j_1 - j_2 - \cdots - j_k - j_1$. Then the assignment $\{(j_1, j_2'), (j_2, j_3'), \ldots, (j_k, j_1'), (j_{k+1}, j_{k+1}'), (j_{k+2}, j_{k+2}'), \ldots, (j_n, j_n')\}$ has a negative cost. Therefore, the cost of the optimal assignment must be negative. Conversely, suppose that the cost of an optimal assignment $\{(j_1, j_2'), (j_2, j_3'), \ldots, (j_k, j_1'), (j_{k+1}, j_{k+1}'), (j_{k+2}, j_{k+2}'), \ldots, (j_n, j_n')\}$ is negative. This solution must contain at least one arc of the form (j_1, j_2') with $j_1 \neq j_2$. If $j_3' = j_1'$, we stop; otherwise, we consider the arc (j_3, j_4'). Repeating this argument as many times as necessary, we eventually find a partial assignment defined as $\{(j_1, j_2'), (j_2, j_3'), \ldots, (j_k, j_1')\}$. The cost of this partial assignment is zero or negative because it can be no more expensive than the partial assignment $\{(j_1, j_1'), (j_2, j_2'), \ldots, (j_k, j_k')\}$. Since the optimal assignment cost is negative, some partial assignment must be negative. But then by the construction of the transformed network, the cycle $j_1 - j_2 - \cdots - j_k - j_1$ is a negative cost cycle in the original network. For our example, the optimal assignment in Figure 12.21(b) is $\{(1, 1'), (3, 3'), (2, 4'), (4, 5'), (5, 2')\}$ and has cost equal to -1. This assignment defines the negative cycle 2–4–5–2 of cost -1 in the original network given in Figure 12.21(a).

If the original network contains no negative cost cycle, we can obtain a shortest path between a specific pair of nodes, say from node 1 to node n, as follows. We consider the transformed network as described earlier and delete the nodes $1'$ and n and the arcs incident to these nodes. [See Figure 12.21(c) for an example of this transformation; in this figure we have modified the cost of arc (2, 4) to 8 so that the network contains no negative cost cycle.] Observe that each path from node 1 to node n in the original network has a corresponding assignment of the same cost in the transformed network, and the converse is also true. For example, the path 1–2–4 in Figure 12.21(a) corresponds to the assignment $\{(1, 2'), (2, 4'), (3, 3'),$

$(5, 5')\}$ in Figure 12.21(c), and the assignment $\{(1, 2'), (2, 4'), (4, 5'), (3, 3')\}$ in Figure 12.21(c) corresponds to the path 1–2–4–5 in Figure 12.21(a). Consequently, an optimal assignment in the transformed network gives a shortest path in the original network.

Shortest Paths in Undirected Networks

Having shown how to transform any shortest path problem (with arbitrary arc costs) in a directed network into an assignment problem, we now study shortest path problems in undirected networks. As we noted in Section 2.4, solving any shortest path problem in an undirected network G with nonnegative arc costs is quite easy; we simply replace each arc (i, j) in the undirected network G with cost c_{ij} by two directed arcs (i, j) and (j, i), both with the same cost c_{ij}, and solve the shortest path problem in the directed network. If the undirected network G contains some arc (i, j) with a negative cost c_{ij}, however, this transformation creates a negative cycle i–j–i. By transforming the undirected problem into a directed problem, we create a negative cycle even though G itself might not contain any negative cycle. Recall from Chapters 4 and 5 that the shortest path algorithms for directed networks do not apply to networks with negative cycles. Consequently, the preceding transformation does not allow us to solve shortest path problems with arbitrary arc lengths on undirected networks. Indeed, solving shortest path problems on undirected networks with negative arc lengths (but with no negative cycles) is substantially harder than the corresponding problem with nonnegative arc lengths; nevertheless, the problem is still solvable in polynomial time. We next describe a transformation that reduces the problem to a minimum weight nonbipartite perfect matching problem.

We perform this transformation in three stages. First, we transform the shortest path problem into a minimum weight perfect b-matching problem. For a given nonnegative n-vector b, we say that a subgraph G' of G is a *perfect b-matching* if each node i has exactly $b(i)$ incident arcs in G'. We illustrate the transformation using the shortest path problem shown in Figure 12.22(a). Suppose that we want to solve the shortest path problem from the source node $s = 1$ to the sink node $t = 4$. As shown in Figure 12.22(b), we add a loop (i, i) of zero cost for each node i, except nodes s and t, and consider the perfect b-matching in the resulting network G' with $b(s) = b(t) = 1$, and $b(i) = 2$ for other nodes. If we observe that each loop arc (i, i) in the matching contributes a degree of 2 units to node i, it is easy to see that any perfect b-matching in G' corresponds to a path in G from node s to node t, and vice versa. [The perfect matching contains the loop arc (i, i) whenever the shortest path from node s to node t does not contain node i.] This observation shows that we can solve the shortest path problem in G by solving the perfect b-matching problem in G'.

We might be tempted, at this point, to try to transform the perfect b-matching problem into a perfect matching problem by splitting those nodes with the degree condition $b(i) = 2$ into two nodes. However, in making this transformation, we encounter one particular difficulty: After splitting the nodes, an arc (i, j) in the original network will correspond to two arcs (i', j) and (i'', j) in the new network that are incident to the copies i' and i'' of the split node i; therefore, the perfect matching in the transformed network might contain both the arcs (i', j) and (i'', j)

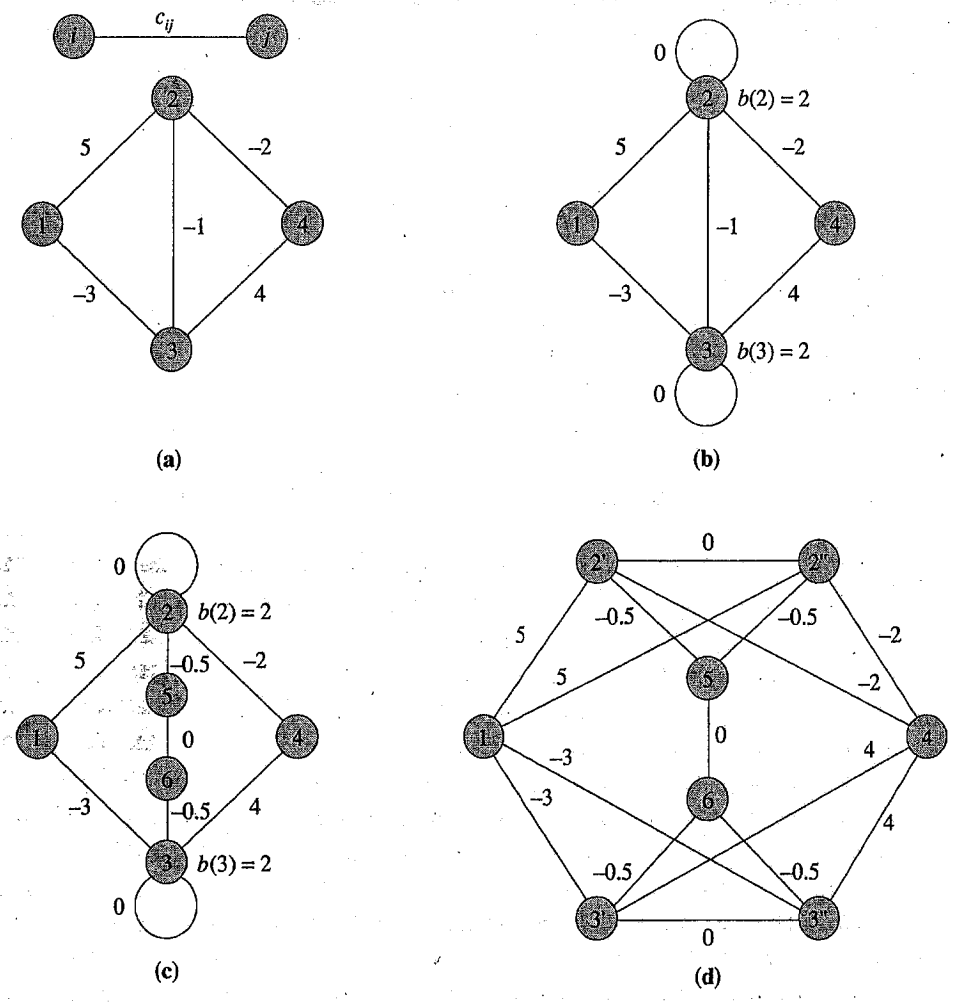


Figure 12.22 Formulating a shortest path problem in an undirected network as a weighted nonbipartite matching problem.

which corresponds to using the original arc (i, j) twice. [We say that we have "used the arc (i, j) twice" when this situation occurs.] To overcome this difficulty, we introduce an additional stage in our transformation.

In the second stage of the transformation, we insert two additional nodes, say nodes k and l, in the middle of each arc (i, j) for which $i \neq j$ and $b(i) = b(j) = 2$, and replace the arc (i, j) by the arcs (i, k), (k, l), and (l, j). We set $b(k) = b(l) = 1$, $c_{ik} = c_{lj} = c_{ij}/2$, and $c_{kl} = 0$. Let G'' denote the graph obtained from this transformation; Figure 12.22(c) shows this graph for our example. To see the equivalence of b-matchings in G' and G'', note that if arc (i, j) is an element of a b-matching M' in G', the corresponding b-matching M'' in G'' would contain the arcs (i, k) and (l, j). Conversely, if arc (i, j) is not contained in the b-matching M', then (k, l) is contained in the b-matching M''. Also, notice that after we have made this transformation, for each arc (p, q) in G'', either $b(p) = 1$ or $b(q) = 1$, which ensures that the matching that we obtain in the subsequent node splitting transformation will use no arc from the graph G'' twice.

In the third stage of the transformation, we construct a third network G''' by splitting each node i with $b(i) = 2$ into two nodes i' and i''; for each arc (i, j) in G'', we introduce two arcs (i', j) and (i'', j) with the same cost as the arc (i, j). Because $b(j) = 1$, any perfect matching of G''' will contain at most one of the arcs (i', j) and (i'', j). For our example, Figure 12.22(d) is the resulting network. It is easy to establish an equivalence between perfect b-matchings in G'' and perfect matchings in G'''. Tracing the steps of these transformations, this result shows that each perfect matching in G''' corresponds to a path from node s to node t in the original graph G; moreover, because the perfect matching and the path have the same cost, we can obtain a shortest path in the undirected graph G by solving the minimum cost perfect matching problem by any polynomial-time algorithm.

12.8 SUMMARY

Matching problems are an important class of optimization models that lie at the interface between network flows and more general problems in combinatorial optimization. The algorithms for certain types of matching problems (those defined on bipartite networks) are streamlined versions of network flow algorithms that we have developed in previous chapters. Although the solution methods for other matching problems (nonbipartite problems) are quite different from those that we have developed for the minimum cost flow problem and its variants, these algorithms do borrow ideas of augmenting paths. Moreover, matching problems are rather intimately related to shortest path problems. In this chapter we studied the following matching problems: (1) the bipartite cardinality matching problem, (2) the bipartite weighted matching problem (also known as the assignment problem), (3) the stable marriage problem, and (4) the nonbipartite cardinality matching problem.

As we have seen, since bipartite matching problems are transformable into network flow problems, we can solve these problems using the algorithms we have developed in previous chapters: for example, we can solve the bipartite cardinality matching problem as a maximum flow problem in unit capacity simple networks (as discussed in Section 8.2). This approach gives an $O(\sqrt{n}m)$ algorithm for solving the problem. In the same way we can transform the weighted bipartite matching problem into a minimum cost flow problem, so we can use the algorithms we have developed in Chapters 9, 10, and 11. The resulting minimum cost flow problem is, however, simpler than general versions of this problem (because the supply/demand vector has only $+1$ or -1 elements), so the minimum cost flow algorithms run faster. By adapting the three pseudopolynomial-time minimum cost flow algorithms discussed in Chapter 9, we have been able to solve the assignment problem in $O(n\ S(n, m, C))$ time, where $S(n, m, C)$ denote the time required to solve a shortest path problem with nonnegative arc lengths. We achieved the best time bound, however, by adapting the cost scaling algorithm: the resulting algorithm runs in $O(\sqrt{n}m \log(nC))$ time.

Nonbipartite matching problems are significantly more difficult to solve. In this chapter we discussed only the cardinality version of this problem; we did not consider weighted nonbipartite matching. Although we can solve the bipartite cardinality matching problem by an augmenting path algorithm, the direct extensions of the algorithm do not solve nonbipartite matching problems. This approach fails because the network might contain blossoms. However, if we contract a blossom whenever

we discover one, we can use an augmenting path algorithm. This approach yields an $O(n^3)$ algorithm for the cardinality matching problem. The proof of the algorithm and its worst-case analysis are intricate and much more difficult than the corresponding analysis for the bipartite cardinality matching problem.

Matching problems are closely related to shortest path problems. Many algorithms for weighted matching problems—such as the successive shortest path, Hungarian and relaxation algorithms—use the shortest path algorithm for problems with nonnegative arcs as a subroutine. Conversely, we can transform shortest path problems with arbitrary arc lengths into matching problems, and these transformations provide some of the best available time bounds for solving shortest path problems. In this chapter we showed how to transform the shortest path problem on directed networks into a bipartite weighted matching problem (i.e., the assignment problem) and how to transform the shortest path problem on undirected networks into a nonbipartite weighted matching problem.

REFERENCE NOTES

Matching problems have received a great deal of attention in the literature. The book by Lovász and Plummer [1986] presents an extensive wealth of information and references on matching theory. In this discussion we cite several key references to the literature, placing an emphasis on theoretically efficient algorithms. Ahuja, Magnanti, and Orlin [1989] present more extensive reference notes on the assignment problem.

Bipartite cardinality matching problems. Hopcroft and Karp [1973] gave an $O(n^{5/2})$ algorithm for this problem. Using similar ideas, Even and Tarjan [1975] obtained an $O(\sqrt{n}m)$ algorithm for the maximum flow problem on unit capacity simple networks. This algorithm, in turn, provides an $O(\sqrt{n}m)$ algorithm for the bipartite cardinality matching problem, which is still the best available time bound for solving this problem.

Nonbipartite cardinality matching problem. The backbone of the nonbipartite matching algorithm is the important characterization result: a matching is optimal if and only if it contains no augmenting path. This theorem is due to Berge [1957], who also gave an exponential time algorithm for identifying an augmenting path. Edmonds [1965a] obtained the first polynomial-time algorithm for this problem, with a time bound of $O(n^4)$. Researchers subsequently developed several improved implementations of this algorithm. Some notable contributions in chronological order are (1) an $O(n^2m)$ algorithm by Witzgall and Zahn [1965], (2) an $O(n^3)$ algorithm by Gabow [1975], (3) an $O(n^{5/2})$ algorithm by Even and Kariv [1975], (4) an $O(nm)$ algorithm by Kameda and Munro [1974], and finally, (5) an $O(n^{1/2}m)$ algorithm by Micali and Vazirani [1980]. The algorithm by Micali and Vazirani is still the fastest available algorithm for solving the nonbipartite cardinality matching problem; its running time is comparable to the running time of the best bipartite cardinality matching algorithm. Vazirani [1989] offered a complete version of this algorithm and its proof. Ball and Derigs [1983] described data structures required for implementing matching algorithms.

Assignment problem. The assignment problem has been a popular, heavily studied research topic within the operations research community. The paper by Ahuja, Magnanti, and Orlin [1989] presented a detailed survey of assignment algorithms. Kuhn [1955] developed the first (primal–dual) algorithm for the assignment problem. Although researchers have developed several different algorithms for the assignment problem, many of these algorithms share common features. The successive shortest path algorithm for the minimum cost flow problem, discussed in Section 9.7, appears to lie at the heart of many (apparently different) assignment algorithms. This approach yields an $O(n\, S(n, m, C))$ time algorithm for solving the assignment problem, where $S(n, m, C)$ is the time needed for solving a shortest path problem with nonnegative arc lengths. Currently, $S(n, m, C) = O(\min\{m + n \log n, m \log\log C, m + n\sqrt{\log C}\})$. Therefore, $O(nm + n^2 \log n)$ is the best available strongly polynomial time bound for solving the assignment problem. Gabow and Tarjan [1989a] developed a cost scaling algorithm for the assignment problem that runs in $O(n^{1/2}m \log(nC))$ time. Bertsekas [1988] proposed an auction algorithm for the assignment problem. Incorporating scaling in the auction algorithm, Orlin and Ahuja [1992] also obtained an $O(n^{1/2}m \log(nC))$ time algorithm; this is the algorithm that we mentioned in Section 12.4. The reference notes for Chapter 11 provide references for simplex-based approaches for the assignment problem. Carpento, Martello, and Toth [1988] presented FORTRAN codes for several algorithms for the assignment problem. For recent computational studies of assignment algorithms, see Bertsekas [1988], Zaki [1990], and Kennington and Wang [1990].

Nonbipartite weighted matching problems. Edmonds [1965b] gave the first algorithm for the nonbipartite weighted matching problem. Gabow [1975] and Lawler [1976] developed $O(n^3)$ implementations of this algorithm. Currently, the fastest algorithms for this problem are (1) an $O(nm + n^2 \log n)$ algorithm due to Gabow [1990], and (2) an $O(m \log(nC)\sqrt{n\alpha(m, n)} \log n)$ algorithm due to Gabow and Tarjan [1989b]. For information concerning the empirical behavior of nonbipartite weighted matching algorithms, see Grötschel and Holland [1985].

Stable marriage problem. Our discussion of this problem has presented the most basic results obtained by Gale and Shapley [1962]. The book by Gusfield and Irving [1989] on the stable marriage problem contains a wealth of information on this topic. The paper by Roth, Rothblum, and Vande Vate [1990] studied polyhedral aspects of the stable marriage problem and used linear programming theory to obtain simpler proofs of many fundamental results for the problem.

Paths and assignments. We presented two transformations to reduce shortest path problems to matching problems. The transformation of the shortest path problem in directed networks to an assignment problem is due to Hoffman and Markowitz [1963] and the transformation of the shortest path problem in undirected networks to the nonbipartite weighted matching problem is due to Edmonds [1967].

The applications of matchings that we gave in Section 12.2 are adapted from the following papers:

1. Bipartite personnel assignment (Machol [1970] and Ewashko and Dudding [1971])
2. Nonbipartite personnel assignment (Meggido and Tamir [1978])
3. Assigning medical graduates to hospitals (Gale and Shapley [1962])
4. Dual completion of oil wells (Devine [1973])
5. Determining chemical bonds (Dewar and Longuet-Higgins [1952])
6. Locating objects in space (Brogan [1989])
7. Matching moving objects (Brogan [1989] and Kolitz [1991])
8. Optimal depletion of inventory (Derman and Klein [1959])
9. Scheduling of parallel machines (Horn [1973])

Elsewhere in the book, we discussed the following applications of matching problems: (1) rewiring of typewriters (Application 1.5, Machol [1961]), (2) pairing stereo speakers (Application 1.6, Mason and Philpott [1988]), (3) the dating problem (Exercise 1.5), (4) the pruned chessboard problem (Exercise 1.6), (5) large-scale personnel assignment (Exercise 1.4), (6) solving shortest path problems in directed and undirected networks (Section 12.7), (7) school bus driver assignment (Exercise 12.1, R. B. Potts), (8) the ski instructor's problem (Exercise 12.2), (9) the undirected Chinese postman problem (Application 19.15, Edmonds and Johnson [1973]), and (10) discrete location problems (Application 19.16, Francis and White [1976]).

Additional applications of the matching problems arise in (1) two-processor scheduling (Fujii, Kasami, and Ninomiya [1969]), (2) determining the rank of a matrix (Anderson [1975]), (3) vehicle and crew scheduling (Carraresi and Gallo [1984]), and (4) making matrices optimally sparse (Hoffman and McCormick [1984]).

EXERCISES

12.1. **School bus driver assignment** (R. B. Potts). A bus company has n morning runs and n afternoon runs that it needs to assign to its n drivers. The runs are of different duration. If the total duration of the morning and afternoon runs assigned to a driver is more than a specified number D, the driver receives a premium payment for each hour of overtime. The company would like to assign the runs to the drivers to minimize the total number of overtime hours.

(a) Formulate this problem as a matching problem.

(b) Suppose that we arrange the morning runs in the nondecreasing order of their duration and the afternoon runs in the nonincreasing order of their duration. Show that if we assign each driver i to the ith morning run and the ith afternoon run, we obtain the optimal assignment.

12.2. **Ski instructor's problem.** A ski instructor needs to assign n pairs of skis to n novice skiers. The skis are available in lengths $l_1 \leq l_2 \leq \cdots \leq l_n$, and the skiers have heights $h_1 \leq h_2 \leq \cdots \leq h_n$. The lengths of the skis assigned to a skier should be proportional to his height: Assume that the constant of proportionality is α. The instructor wishes to assign the skis to skiers so that the total difference between the actual ski lengths and the ideal ski lengths is as small as possible. Show that if for each i, she assigns the ith skier to the ith pair of skis, her assignment is optimal.

12.3. A budding connoisseur plans to consume one of n bottles of wine in his cellar on Saturday evening for each of the next n weeks. The age a_i of bottle i is known (in weeks). The utility of bottle i as a function of time t (in weeks) is given by $b_i t^3 - c_i t$, for some constants b_i and c_i. The connoisseur wants to know how he should consume his wine to maximize the total utility of wine he consumes. Formulate this problem as an assignment problem using the following data.

i	1	2	3	4	5	6
a_i	10	5	4	20	10	15
b_i	2	3	5	3	1	4
c_i	10	15	20	5	25	15

12.4. Show how to solve the bin packing problem described in Exercise 3.10 as a matching problem if $a_j > \frac{1}{3}$ for each $j = 1, \ldots, n$.

12.5. When the utility function for items has a special form, we can solve the optimal inventory depletion problem discussed in Application 12.8 very efficiently.

(a) Show that when the utility function $v(t)$ is a concave function, the optimal policy is to issue the youngest item first.

(b) Show that when the utility function $v(t)$ is a convex function, the optimal policy is to issue the oldest item first.

12.6. This exercise develops a justification for the assignment formulation of the machine scheduling problem discussed in Application 12.9. Let us refer to a feasible assignment as a *proper assignment* if it assigns the jobs to each machine in consecutive places, including the last place. For instance, if we assign jobs 1, 4, and 5 to some machine, assigning job 5 to the last place, job 1 to the second to last place, and job 4 to the fifth to last place, the resulting assignment is not a proper assignment.

(a) Show that we can always improve a nonproper assignment. Conclude that any optimal assignment of the assignment problem is a proper assignment.

(b) Establish a one-to-one correspondence, which preserves costs, between feasible schedules of the scheduling problem and proper assignments of the assignment problem. Conclude that the solution of the assignment problem will yield an optimal schedule for the scheduling problem.

12.7. Determine a maximum cardinality matching in the graph shown in Figure 12.4.

12.8. Let M_1 and M_2 be two arbitrary matchings in a bipartite network $G = (N_1 \cup N_2, A)$. Show that some matching M matches all the nodes in N_1 that are matched by M_1 and all the nodes of N_2 that are matched in M_2. (*Hint*: Consider $M_1 \oplus M_2$ and modify the matching M_1 or M_2 appropriately.)

12.9. The army would like to transfer five servicemen to five new posts in a way that minimizes the total moving cost. The accompanying table specifies allowable assignments and the moving cost for each possible assignment. Use the relaxation algorithm to determine an assignment that minimizes the total moving cost.

Serviceman \ Posting	1	2	3	4	5
1	25	30	—	—	—
2	20	—	70	35	—
3	80	75	90	65	—
4	—	—	—	55	40
5	—	—	—	60	50

12.10. A construction company needs to assign four workers to four jobs. The accompanying table specifies the workers' proficiency scores for the jobs. A dash "–" in position (i, j) indicates that worker i is unqualified to perform job j. Use the successive shortest path algorithm to identify an assignment that maximizes the total proficiency scores for carrying out the jobs.

Worker \ Job	1	2	3	4
1	45	—	—	30
2	50	55	15	—
3	—	60	25	75
4	45	—	—	35

12.11. Let $G = (N_1 \cup N_2, A)$ be a bipartite network with $| N_1 | = | N_2 | = n_1$. For any set $S \subseteq N_1$, define *neighbor(S)* as the set of nodes in N_2 that are adjacent to the nodes in S.

(a) Show that if G has a perfect matching, then for any subset $S \subseteq N_1$, $|$ neighbor(S) $| \geq | S |$.

(b) Show that if for every subset $S \subseteq N_1$, $|$ neighbor(S) $| \geq | S |$, then G has a perfect matching. Conclude that G has a perfect matching if and only if for every subset $S \subseteq N_1$, $|$ neighbor(S) $| \geq | S |$. (*Hint*: Prove that using the bipartite matching algorithm given in Figure 12.9, we would either find an augmenting path from every unassigned node in N_1 to a node in N_2, or we would contradict the assumption that $|$ neighbor(S) $| \geq | S |$.)

12.12. Dancing problem. At a high school party attended by n boys and n girls, each boy knows exactly k ($1 \le k \le n$) girls and each girl knows exactly k boys. Assume that the acquaintanceship is mutual (i.e., if b knows c, then c also knows b).

(a) Show that it is always possible to arrange a dance in which each boy dances with a girl he knows. (*Hint*: Use the result of Exercise 12.11.)

(b) Show that it is possible to arrange k consecutive parties so that at each party each boy dances with a different girl that he knows.

12.13. A 0–1 matrix of size $n \times n$ is a *permutation matrix* if each row and column contains a single 1 entry. Let H be a 0–1 matrix of size $n \times n$ and suppose that each row and each column contains exactly k 1's. Show that it is possible to represent H as the sum of k permutation matrices of size $n \times n$. (*Hint*: Use the results in Exercise 12.11 or 12.12.)

12.14. An $n \times n$ matrix R is *doubly stochastic* if all of its elements r_{ij} are nonnegative and if the sum of the elements in each row and each column equals 1. Show that it is possible to represent a doubly stochastic matrix as a convex combination of the permutation matrices (i.e., $R = \alpha_1 P_1 + \alpha_2 P_2 + \cdots + \alpha_p P_p$ for some set of permutation matrices $P_1, P_2, \ldots, P_p$ and some positive weights α_j satisfying the condition $\alpha_1 + \alpha_2 + \cdots + \alpha_p = 1$). Moreover, show that we can choose the permutation matrices so that p is no more than the number of nonzero elements in the matrix R. (*Hint*: Let $Q = \{q_{ij}\}$ be a 0–1 matrix with $q_{ij} = 1$ if r_{ij} is positive. Show that Q contains a permutation matrix. Modify R and use this result repeatedly.)

12.15. Suppose that each of 10,000 individuals serves on exactly 13 of 10,000 committees and each committee has exactly 13 members. Show that we can order the committees so that for each $j = 1, \ldots, 10{,}000$, individual j serves on committee j. (*Hint*: Use the result of some previous exercise.)

12.16. An *arc coloring* (or, simply, a coloring) of an undirected graph $G = (N, A)$ is a coloring of the arcs with several colors so that no two arcs incident to the same node have the same color. A k-coloring is a coloring of the arcs with k distinct colors.

(a) Show that a k-coloring of the graph G decomposes the arcs in A into k arc-disjoint matchings.

(b) A graph is *k-regular* if the degree of each node is exactly k. Show that a k-regular bipartite graph has a k-coloring.

(c) Let $\delta(G)$ denote the maximum degree of any node in the graph G. Show that every bipartite graph G has a $\delta(G)$ coloring. (*Hint*: Show how to make the graph $\delta(G)$-regular by adding additional nodes and arcs.)

12.17. A small manufacturing company produces a speciality home security device composed of p components. The company employs p individuals who have different expertise; each of them must spend time in the production of each component: the ith worker must spend a_{ij} hours (an integer) on component j. The company wants to determine the minimum number of hours required to produce the device so that (1) no worker is working on two different components at the same time, and (2) no more than one person is working on any component at any one time.

(a) For each $1 \le i \le p$, let $r_i = \sum_{j=1}^{p} a_{ij}$, and for each $1 \le j \le p$, let $c_j = \sum_{i=1}^{p} a_{ij}$. Also let α be the largest value of all the parameters r_i and c_j. Show that the company requires at least α hours to produce the device.

(b) Use the result in part (a) to show that we can increase some of the a_{ij}'s so that each r_i and each c_j equals α.

(c) Use the results in part (b) and Exercise 12.14 to show that the company can always produce the device in α hours.

12.18. In this chapter we considered the assignment problem on bipartite networks $G = (N_1 \cup N_2, A)$ with $|N_1| = |N_2|$. Consider a modified version of the assignment problem when $|N_1| < |N_2|$. In this case, in a feasible assignment, we require all the nodes in N_1 to be matched, but permit some of the nodes in N_2 to be unmatched.

Show that we can transform this modified assignment problem to the (original) assignment problem.

12.19. Show how to transform the (uncapacitated) transportation problem into an assignment problem on an expanded network. Next show how to transform the (capacitated) minimum cost flow problem into an assignment problem. Justify your transformation. (*Hint*: If U denotes the magnitude of the largest supply/demand at a node, transforming the transportation problem to an assignment problem yields a network with $O(nU)$ nodes.)

12.20. Consider an assignment problem whose arc costs are all 0 or 1. Which assignment algorithm discussed in this chapter would solve this problem in the least possible time?

12.21. The president's office at a university needs to assign a targeted group of n faculty to be chairs of n committees. Each person proposes, in decreasing order of preference, a list of three committees that he or she would like to chair. We want to determine whether we could possibly find a *satisfiable assignment* (i.e., one that assigns the faculty to the committees so that each faculty member obtains a job on his or her list). If some satisfiable assignment is possible, we want to find the assignment that maximizes the number of faculty with their most preferred committee chair, and further, among such assignments, the assignment that maximizes the number of faculty with their second most preferred committee chair. Show how to solve this problem by solving a single assignment problem. (*Hint*: Assign arc costs appropriately.)

12.22. Factored assignment problems

(a) A factored assignment problem is an assignment problem on a complete bipartite network $G = (N_1 \cup N_2, A)$ (i.e., $A = N_1 \times N_2$) when each arc cost is specified as $c_{ij} = \alpha_i \alpha_j$ for some set of node numbers α_i associated with the nodes in $N_1 \cup N_2$. Assume that we are given the node numbers α_i's sorted in nondecreasing order for each of the sets N_1 and N_2. [Notice that the input size of the factored assignment problem is only $O(n)$.] Describe an $O(n)$ algorithm for solving this specialized version of the assignment problem. (*Hint*: First solve the problem when $|N_1| = 2$ and then generalize your result.)

(b) Consider the assignment problem on a complete bipartite network $G = (N_1 \cup N_2, A)$ when the cost data c_{ij} is of the form $c_{ij} = \alpha_i + \alpha_j$ for some node numbers α_i associated with the nodes in $N_1 \cup N_2$. Describe an $O(n)$ algorithm for solving this assignment problem.

12.23. Bottleneck assignment problem. The bottleneck assignment problem is an important variation of the classical assignment problem that arises in the following scenario. Suppose that an assembly line has p jobs to be assigned to p operators. Let c_{ij} denote the number of units per hour that operator i can process if assigned to job j. For a given assignment, the output rate of the assembly line is given by the minimum c_{ij} in the assignment, which we would like to maximize over all assignments. In the bottleneck assignment problem we wish to determine an assignment for which the least costly assignment is as large as possible. This is a *maximin* version of the bottleneck assignment problem; similarly, we could define the *minimax* version in which we want to determine an assignment for which the most costly assignment is as small as possible. In the following exercise, we consider the minimax version of the problem.

Suppose we sort the arc costs c_{ij}'s: let $c_1 < c_2 < \cdots < c_k$ denote the sorted list of distinct values of these costs ($k \leq m$). Let FS(l, M) denote a subroutine that for any input number $l \geq 1$, determines whether in some assignment every arc cost is no more than c_l. If no such assignment exists, M is a null set.

(a) Describe an $O(\sqrt{n}\, m)$ algorithm for implementing the subroutine FS(l, M).

(b) Show how to solve the bottleneck assignment problem by calling the subroutine FS(l, M) $O(\log k)$ times.

12.24. Balanced assignment problem (Martello et al. [1984]). The balanced assignment problem is another variation of the classical assignment problem, which is perhaps best illus-

trated by some specific scenarios. Given n people and n tasks, let c_{ij} denote the amount of work person i requires to perform job j. Suppose that we are interested in choosing a pairing of workers and jobs that distributes the work load as evenly as possible. As another problem setting for the balanced assignment problem, we let c_{ij} be the expected lifetime of component j produced by a company i; we wish to choose the components so that we will need to replace all the components at about the same time. In these settings and in general, in the balanced assignment we wish to determine an assignment that will minimize the difference between the most costly and the least costly assignment. In this exercise we develop an algorithm for solving this version of the assignment problem.

Suppose that we sort the arc costs c_{ij}'s and let $c_1 < c_2 < \cdots < c_k$ denote the sorted list of distinct values of these costs ($k \leq m$). Let FS(l, u, M) denote a subroutine that takes as an input two numbers l and u, satisfying the condition $1 \leq l \leq u \leq k$, and determines whether in some assignment M every arc cost is between c_l and c_u; if no such assignment exists, then M is a null set.

(a) Describe an $O(\sqrt{n}\, m)$ algorithm for implementing the subroutine FS(l, u, M).
(b) Show how to solve the balanced assignment problem by calling the subroutine FS(l, u, M) $O(k)$ times.

12.25. Solve the stable marriage problem shown in Figure 12.23 when the matrix M gives the rankings of men for women and matrix W gives the ranking of women for men. A higher ranking implies a greater preference.

M

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

W

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

Figure 12.23 Men and women rankings.

12.26. Give a 2 × 2 example of the stable marriage problem (i.e., two men and two women) which has at least two distinct stable matchings.

12.27. Show that in a man-optimal matching, each woman has the least preferred partner that she can have in any stable matching.

12.28. Suppose that in the stable marriage problem, men and women might prefer to remain unmarried if they do not find suitable mates. In this version of the problem, each man and each woman makes a list of potential mates in order of preference but does not list anyone whom he/she is unwilling to marry. Now a stable marriage will permit some men/women to remain unmarried. Show how to modify the stable matching algorithm given in Section 12.5 to find such a matching. Suppose that a man i is unmarried in the matching produced by your algorithm. Is it true that he is unmarried in all stable marriages? Prove your answer.

12.29. In Section 12.5 we described an algorithm for constructing a man-optimal matching. Modify this algorithm so that it constructs a woman-optimal matching. Apply this algorithm to the example given in Figure 12.23. Is the woman-optimal matching different than the man-optimal matching?

12.30. Suppose that you are given a matrix denoting men's ranking of women: a higher rank denotes a more favored woman. Assume that each rank is an integer between 1 and n. Using this matrix, you wish to prepare the *priority list* of each man, which lists the women in nonincreasing order of their rankings. Show how you can construct priority lists of all men in a total of $O(n^2)$ time.

12.31. **Stable university admissions.** In the stable marriage algorithm, each man is matched to one woman and each woman is matched to one man. Generalize this algorithm to find a stable assignment of medical school graduates to hospitals (see Application 12.3). In this case each hospital i can hire a_i graduates.

12.32. **Unstable roommates** (Gale and Shapley [1962]). The headmaster of a boarding school needs to divide an even number of boys into pairs of roommates. In this setting, a set of pairings is *stable* if no two boys who are not roommates prefer each other to their actual roommates. Show that in some situations no stable pairing is possible. (*Hint*: There is an example with four boys.)

12.33. Consider the graph shown in Figure 12.24(a) with a matching shown by the bold lines. In this graph, specify (1) an alternating path of length 10; (2) an alternating cycle of length 10; (3) an augmenting path of length 5; (4) an augmenting path of length 9; and (5) an alternating tree rooted at node 2 that spans all the nodes.

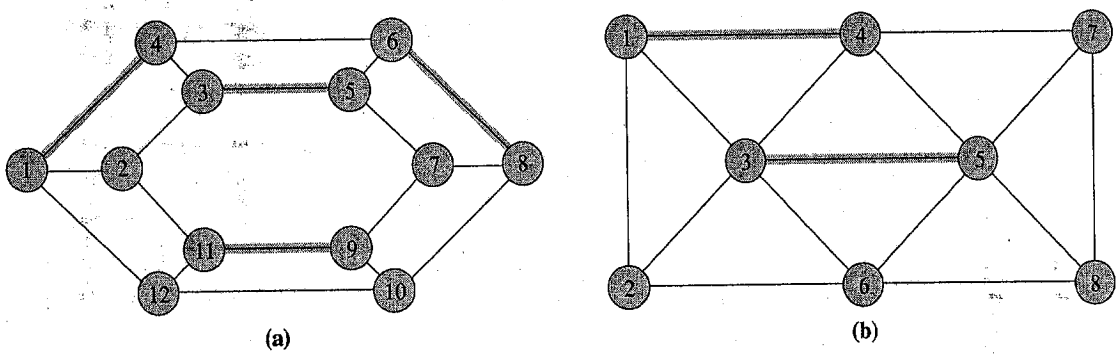


Figure 12.24 Examples for Exercises 12.33 to 12.35.

12.34. Consider the graph shown in Figure 12.24(b) with a matching as shown by the bold lines. Specify a blossom in the graph whose stem contains two arcs. Contract this blossom and show the contracted graph. List all augmenting paths of length 3 in the contracted graph and the corresponding paths in the original (uncontracted) graph.

12.35. Apply the nonbipartite cardinality matching algorithm to the example shown in Figure 12.24(b). Use the matching shown by bold lines as the starting matching.

12.36. Professor May B. Wright has posed the following problem for you to resolve: Consider the graph shown in Figure 12.25(a) which has an augmenting path 1–2–3–4. The se-

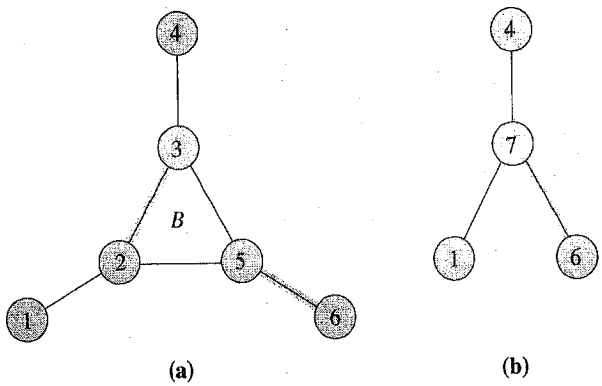


Figure 12.25 Example for Exercise 12.36.

quence of nodes 2–3–5–2 defines a blossom in the graph; contracting the nodes, we obtain the graph shown in Figure 12.25(b). However, the contracted graph does not have any augmenting path. Professor Wright believes that this example contradicts the theorem that the graph has an augmenting path if and only if the contracted graph has an augmenting path. Is she correct?

12.37. Show that a network is nonbipartite if during the application of the search procedure of the nonbipartite matching algorithm (described in Section 12.6), it attempts to assign an odd label to an even-labeled node or an even label to an odd-labeled node. Show that the "only if" part of the result is not true. (*Hint*: For the second part, use the network shown in Figure 12.25.)

12.38. A subset $N' \subseteq N$ is *matchable* if each node in N' is matched in some matching. Let S be a matchable set of nodes. Show that each node in S is matched in some maximum cardinality matching.

12.39. **Berge's theorem** (Berge [1957]). Prove the following theorem from first principles: A matching M^* in a graph G is maximum if and only if the graph G contains no augmenting path with respect to M^*.

12.40. A matching M is a *maximal matching* of $G = (N, A)$ if for every arc $(i, j) \notin M$, $M \cup \{(i, j)\}$ is not a matching.

(a) Show how to construct a maximal matching in $O(m)$ time.

(b) Show that a maximal matching contains at least 50 percent as many arcs as a maximum matching.

12.41. Let $G = (N, A)$ be a graph. Let $\mu(G)$ denote the maximum cardinality of any matching in G. A subset $A' \subseteq A$ is an *arc cover* if for every node $i \in N$, i is an endpoint of at least one arc in A'. Let $\alpha(G)$ denote the minimum cardinality of any arc cover of G. Show that $\mu(G) + \alpha(G) = n$. (*Hint*: Show how to extend any matching of cardinality k into an arc cover of cardinality $n - k$.)

12.42. Suppose that an undirected graph $G = (N, A)$ has a perfect matching. An arc (i, j) in G is *unmatchable* if no perfect matching contains the arc (i, j).

(a) Let G^{ij} denote the graph obtained by deleting from G the nodes i and j and the arcs incident to these nodes. Show that an arc (i, j) is unmatchable if and only if G^{ij} has no perfect matching. Use this result to develop a polynomial-time algorithm for identifying all unmatchable arcs.

(b) Specify an $O(n^3)$ algorithm for finding all unmatchable arcs in a graph G. (*Hint*: First find a perfect matching. Then show how to find in $O(m)$ time all unmatchable arcs incident to any node i.)

12.43. Consider a nonbipartite graph G that has a perfect matching. In general, we might expect that a maximum weight matching of G would be perfect. Show that this is not always true by constructing an example in which a maximum weight matching is not a perfect matching. (*Hint*: Try an example with four nodes.)

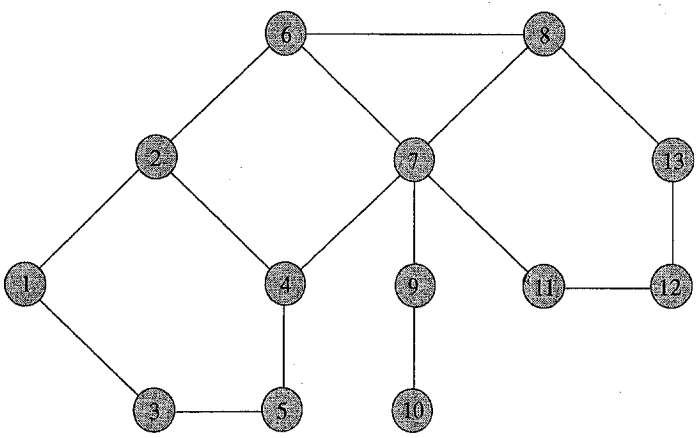


Figure 12.26 Policemen's problem.

12.44. Describe a polynomial-time algorithm for solving the maximum weight matching problem on a tree.

12.45. **Policemen's problem** (Gondran and Minoux [1984]). Consider an undirected graph shown in Figure 12.26, which represents the street map of a town. A policeman standing in the middle of a street can survey the crossings at the two ends of the street. What is the smallest number of policemen needed to survey all the crossings in the network? (*Hint*: Use the result of a previous exercise.)

13

MINIMUM SPANNING TREES

I think I will never see,
A poem as lovely as a tree.
—Joyce Kilmer

Chapter Outline

13.1 INTRODUCTION

As we have seen repeatedly throughout earlier chapters, spanning trees play a central role within the field of network flows. In solving the shortest path problem in Chapters 4 and 5, we constructed (shortest path) spanning trees rooted at a source node. The simplex method for solving minimum cost flow problems that we discussed in Chapter 11 is a spanning tree manipulation algorithm that iteratively moves from one spanning tree to another, at each step introducing one arc into the spanning tree in place of another. As we have also seen in Chapter 11 (Theorem 11.3), minimum cost flow problems always have spanning tree solutions; therefore, in principle to solve any minimum cost network flow problem, including shortest path problems and maximum flow problems, we can always restrict our attention to spanning tree solutions. Since any network has only a finite number of spanning trees, we can view any network flow problem as a discrete optimization model and solve it in a finite number of iterations.

In this chapter we consider another spanning tree model, known as the *minimum spanning tree problem*. Recall that a spanning tree T of G is a connected acyclic subgraph that spans all the nodes. Every spanning tree of G has $n - 1$ arcs (see Property 2.2). Given an undirected graph $G = (N, A)$ with $n = |N|$ nodes and $m = |A|$ arcs and with a *length* or *cost* c_{ij} associated with each arc $(i, j) \in A$, we wish to find a spanning tree, called a *minimum spanning tree*, that has the smallest total cost (or length) of its constituent arcs, measured as the sum of costs of the arcs

in the spanning tree. Note that minimal spanning trees differ from the shortest path tree that we have considered in Chapters 4 and 5 in the following two respects:

1. For the minimum spanning tree problem, the arcs are undirected. [Since the network is undirected, we refer to the arc between the node pair i and j as either (i, j) or (j, i).] For the version of the shortest path problems that we considered previously, the networks were directed. This distinction is unimportant in one sense: We could easily have developed our prior results for shortest path problems using undirected graphs as well as directed graphs (see Section 2.4). Viewed in another way, however, this distinction is important: Finding a minimum spanning tree on a directed network with all paths directed away from a given root node (this structure is known as a *rooted arborescence*) is a much more difficult problem than the undirected minimum spanning tree problem.
2. Our objective functions for the minimum spanning tree problem and for the shortest path tree problem are quite different. For the minimal spanning tree problem, we count the cost of each arc exactly once; for the shortest path tree problem, we typically count the cost of some arcs several times: equal to the number of paths from the root node that pass through that arc (i.e., the number of shortest paths in the tree that contain that arc).

The minimum spanning tree problem arises in a number of applications, both as a stand-alone problem and as a subproblem in a more complex problem setting. We begin this chapter by describing several such applications. We next consider combinatorially based optimality conditions for assessing whether a given spanning tree is a minimum spanning tree. We consider two such optimality conditions. The first condition is based on comparing the cost of any tree arc with the other arcs contained in the cut defined by removing that arc from the tree. The other is based on comparing the cost of a nontree arc with the tree arcs in the path that connects the endpoints of the nontree arc. These two *cut and path optimality conditions* are easy to state and to develop, yet they quite naturally motivate several algorithms for solving the minimum spanning tree problem.

The resulting algorithms are all very simple, although implementing them efficiently requires considerable care and ingenuity. The three algorithms we consider in this chapter—Kruskal's algorithm, Prim's algorithm, and Sollin's algorithm—all share one characteristic: They are "greedy" algorithms in the sense that at each step they add an arc of minimum cost from a candidate list, as long as the added arc does not form a cycle with the arcs already chosen. All three algorithms maintain a forest containing arcs already chosen and then they add one or more arcs to enlarge the size of the forest. For Kruskal's algorithm, the candidate list is the entire network; for Prim's algorithm, the forest is a single tree plus a set of isolated nodes and the candidate list contains all the arcs between the single tree and the nodes not in the tree; Sollin's algorithm is a hybrid approach that maintains several components in the forest, as in Kruskal's algorithm, but then adds several arcs at each iteration, choosing (like Prim's algorithm) the minimum cost arc connecting each component of the forest to the nodes not in that component.

Since greedy algorithms, such as Kruskal's, Prim's, and Sollin's, arise in many

other problem contexts in discrete optimization, in Section 13.7 we show how a generalization of Kruskal's algorithm will solve a broad class of abstract combinatorial optimization problems known as matroid optimization problems. This discussion not only permits us to show how to solve a new class of combinatorial optimization problems, but also provides additional insight concerning the combinatorial structure of spanning trees that underlies the validity of the greedy solution approach.

Mathematical programming has another useful way to view the minimal spanning tree problem. In Section 13.8 we formulate the minimal spanning tree problem as an integer programming model and use linear programming arguments to establish yet another proof of the validity of Kruskal's algorithm. This discussion serves several purposes: (1) it gives another useful view of minimum spanning trees; (2) it illustrates a proof technique, via linear programming, that has proven to be very powerful in the field of combinatorial optimization; and (3) it provides a bridge between the minimum spanning tree problem and an important topic in discrete optimization, polyhedral combinatorics (i.e., the study of integer polyhedra).

In closing this section we might note that we can also define and study the maximum spanning tree problem, which as its name implies, seeks the spanning tree with the largest total costs of its constituent arcs. Since we can find a maximum spanning tree by multiplying all the arc costs by -1 and then solving a minimum spanning tree, the algorithms and theory of the maximum spanning tree problem are essentially the same as those of the minimum spanning tree problem.

13.2 APPLICATIONS

Minimum spanning tree problems generally arise in one of two ways, directly or indirectly. In some *direct* applications, we wish to connect a set of points using the least cost or least length collection of arcs. Frequently, the points represent physical entities such as components of a computer chip, or users of a system who need to be connected to each other or to a central service such as a central processor in a computer system. In *indirect* applications, we either (1) wish to connect some set of points using a measure of performance that on the surface bears little resemblance to the minimum spanning tree objective (sum of arc costs), or (2) the problem itself bears little resemblance to an "optimal tree" problem—in these instances, we often need to be creative in modeling the problem so that it becomes a minimum spanning tree problem. In this section we consider several direct and indirect applications.

Application 13.1 Designing Physical Systems

The design of physical systems can be a complex task involving an interplay between performance objectives (such as throughput and reliability), design costs and operating economics, and available technology. In many settings, the major criterion is fairly simple: We need to design a network that will connect geographically dispersed system components or that will provide the infrastructure needed for users to communicate with each other. In many of these settings, the system need not have any redundancy, so we are interested in the simplest possible connection, namely, a spanning tree. This type of application arises in the construction (or in-

stallation) of numerous physical systems: highways, computer networks, leased-line telephone networks, railroads, cable television lines, and high-voltage electrical power transmission lines. For example, this type of minimum spanning tree problem arises in the following problem settings:

1. Connect terminals in cabling the panels of electrical equipment. How should we wire the terminals to use the least possible length of the wire?
2. Constructing a pipeline network to connect a number of towns using the smallest possible total length of pipeline.
3. Linking isolated villages in a remote region, which are connected by roads but not yet by telephone service. In this instance we wish to determine along which stretches of roads we should place telephone lines, using the minimum possible total miles of the lines, to link every pair of villages.
4. Constructing a digital computer system, composed of high-frequency circuitry, when it is important to minimize the length of wires between different components to reduce both capacitance and delay line effects. Since all components must be connected, we obtain a spanning tree problem.
5. Connecting a number of computer sites by high-speed lines. Each line is available for leasing at a certain monthly cost, and we wish to determine a configuration that connects all the sites at the least possible cost.

Each of these applications is a direct application of the minimum spanning tree problem. We next describe several indirect applications.

Application 13.2 Optimal Message Passing

An intelligence service has n agents in a nonfriendly country. Each agent knows some of the other agents and has in place procedures for arranging a rendezvous with anyone he knows. For each such possible rendezvous, say between agent i and agent j, any message passed between these agents will fall into hostile hands with a certain probability p_{ij}. The group leader wants to transmit a confidential message among all the agents while minimizing the total probability that the message is intercepted.

If we represent the agents by nodes, and each possible rendezvous by an arc, then in the resulting graph G we would like to identify a spanning tree T that minimizes the probability of interception given by the expression $\{1 - \Pi_{(i,j)\in T}(1 - p_{ij})\}$. Alternatively, we would like to find a tree T that maximizes $\Pi_{(i,j)\in T}(1 - p_{ij})$. We can identify such a tree by defining the length of an arc (i, j) as $\log(1 - p_{ij})$ and solving a maximum spanning tree problem.

Application 13.3 All-Pairs Minimax Path Problem

The minimax path problem is a variant of the maximum capacity path problem that we discussed in Exercise 4.37. In a network $G = (N, A)$ with arc costs c_{ij}, we define the *value* of a path P from node k to node l as the maximum cost arc in P. The all-pairs minimax path problem requires that we determine, for every pair $[k, l]$ of nodes, a minimum value path from node k to node l. We show how to solve the all-pairs

minimax path problem on an undirected graph by solving a single minimum spanning tree problem.

The minimax path problem arises in a variety of situations. As an example, consider a spacecraft that is about to enter the earth's atmosphere. The craft passes through different pressure and temperature zones that we can represent by arcs of a network. It needs to fly along a trajectory that will bring the craft to the surface of the earth while keeping the maximum temperature to which the surface of the craft is exposed as low as possible. As an alternative, we might wish to select a path that will minimize the maximum deceleration during the descent. Other examples of the minimax path problem arise when (1) in traveling through a desert, we want to minimize the length of the longest stretch between rest areas; and (2) in traveling in a wheelchair, a person might wish to minimize the maximum ascent along the path segments.

To transform the all-pairs minimax path problem into a minimum spanning tree problem, let T^* be a minimum spanning tree of G. Let P denote the unique path in T^* between a node pair $[p, q]$ and let (i, j) denote the maximum cost arc in P. Observe that the value of the path P is c_{ij}. By deleting arc (i,j) from T^*, we partition the node set N into two subsets and therefore define a cut $[S, \bar{S}]$ with $i \in S$ and $j \in \bar{S}$ (see Figure 13.1). We later show in Theorem 13.1 that this cut satisfies the following property:

$$c_{ij} \leq c_{kl} \qquad \text{for each arc } (k, l) \in [S, \bar{S}], \tag{13.1}$$

for otherwise by replacing the arc (i, j) by an arc (k, l) we can obtain a spanning tree of smaller cost. Now, consider any path P' from node p to node q. This path must contain at least one arc (k, l) in $[S, \bar{S}]$. Property (13.1) implies that the value of the path P' will be at least c_{ij}. Since c_{ij} is the value of the path P, P must be a minimum value path from node p to node q. This observation establishes the fact that the unique path between any pair of nodes in T^* is the minimum value path between that pair of nodes.

Application 13.4 Reducing Data Storage

In several different application contexts, we wish to store data specified in the form of a two-dimensional array more efficiently than storing all the elements of the array

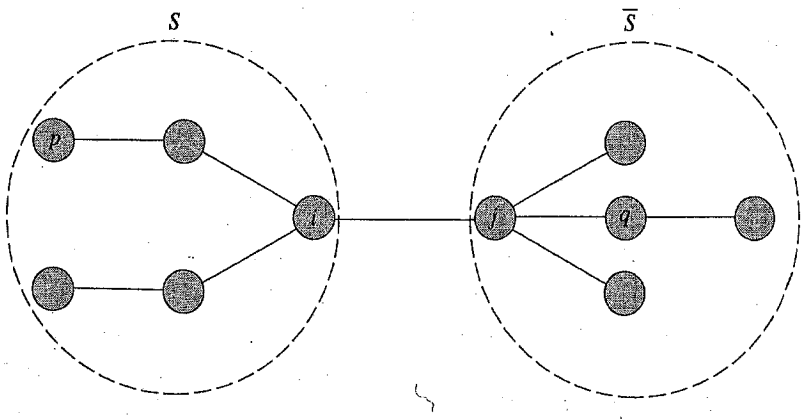


Figure 13.1 Cut formed by deleting the arc (i, j) from a spanning tree.

(to save memory space). We assume that the rows of the array have many similar entries and differ only at a few places. One such situation arises in the sequence of amino acids in a protein found in the mitochondria of different animals and higher plants.

Since the entities in the rows are similar, one approach for saving memory is to store one row, called the *reference row*, completely, and to store only the differences between some of the rows so that we can derive each row from these differences and the reference row. Let c_{ij} denote the number of different entries in rows i and j; that is, if we are given row i, then by making c_{ij} changes to the entries in this row we can obtain row j, and vice versa. Suppose that the array contains four rows, represented by R_1, R_2, R_3, and R_4, and we decide to treat R_1 as a reference row. Then one plausible solution is to store the differences between R_1 and R_2, R_2 and R_4, and R_1 and R_3. Clearly, from this solution, we can obtain rows R_2 and R_3 by making c_{12} and c_{13} changes to the elements in row R_1. Having obtained row R_2, we can make c_{24} changes to the elements of this row to obtain R_4.

It is easy to see that it is sufficient to store differences between those rows that correspond to arcs of a spanning tree. These differences permit us to obtain each row from the reference row. The total storage requirement for a particular storage scheme will be the length of the reference row (which we can take as the row with the least amount of data) plus the sum of the differences between the rows. Therefore, a minimum spanning tree would provide the least cost storage scheme.

Application 13.5 Cluster Analysis

The essential issue in cluster analysis is to partition a set of data into "natural groups"; the data points within a particular group of data, or a cluster, should be more "closely related" to each other than the data points not in that cluster. Cluster analysis is important in a variety of disciplines that rely on empirical investigations. Consider, for example, an instance of a cluster analysis arising in medicine. Suppose that we have data on a set of 350 patients, measured with respect to 18 symptoms. Suppose, further, that a doctor has diagnosed all of these patients as having the same disease, which is not well understood. The doctor would like to know if he can develop a better understanding of this disease by categorizing the symptoms into smaller groupings that can be detected through cluster analysis. Doing so might permit the doctor to find more natural disease categories to replace or subdivide the original disease.

In this section we describe the use of spanning tree problems to solve a class of problems that arise in the context of cluster analysis. Suppose that we are interested in finding a partition of a set of n points in two-dimensional Euclidean space into clusters. A popular method for solving this problem is by using Kruskal's algorithm for solving the minimum spanning tree problem (we describe this method in Section 13.4). As we will show, at each intermediate iteration, Kruskal's algorithm maintains a forest (i.e., a collection of node-disjoint trees) and adds arcs in nondecreasing order of their lengths. We can regard the nodes spanned by the trees at intermediate steps as different clusters. These clusters are often excellent solutions for the clustering problem, and moreover, we can obtain them very efficiently. Kruskal's algorithm can be thought of as providing n partitions: The first partition contains

n clusters, each cluster containing a single point, and the last partition contains just one cluster containing all the points. Alternatively, we can obtain n partitions by starting with a minimum spanning tree and deleting tree arcs one by one in nonincreasing order of their lengths. We illustrate the latter approach using an example. Consider a set of 27 points shown in Figure 13.2(a). Suppose that the network in Figure 13.2(b) is a minimum spanning tree for these points. Deleting the three largest length arcs from the minimum spanning tree gives a partition with four clusters shown in Figure 13.2(c).

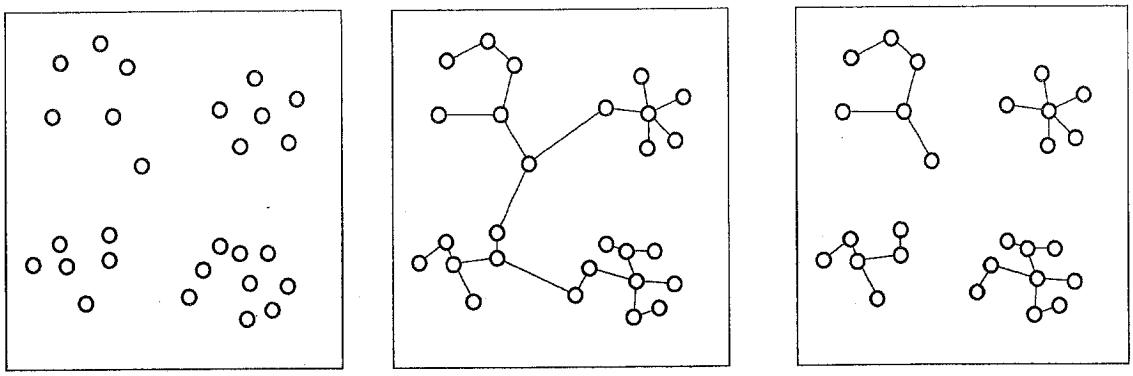


Figure 13.2 Identifying clusters by finding a minimum spanning tree.

Analysts can use the information obtained from the preceding analysis in several ways. The procedure we have described yields n partitions. Out of these, we might select the "best" partition by simple visualization or by defining an appropriate objective function value. A good choice of the objective function depends on the underlying features of the particular clustering application. We might note that this analysis is not limited to points in two-dimensional space; we can easily extend it to multidimensional space if we define interpoint distances appropriately.

13.3 OPTIMALITY CONDITIONS

As in our earlier discussion of network flow algorithms, optimality conditions for the minimum spanning tree problem play a central role in developing algorithms and establishing their validity. For the minimum spanning tree problem, we can formulate the optimality conditions in two important ways: *cut optimality conditions* and *path optimality conditions*. Needless to say, both optimality conditions are equivalent. Before considering these conditions, let us establish some further notation and illustrate some basic concepts.

The subgraphs shown in Figures 13.3(b) and 13.3(c) are spanning trees for the network shown in Figure 13.3(a). However, the subgraph shown in Figure 13.3(d) is not a spanning tree because it is not connected, and the subgraph shown in Figure 13.3(e) is not a spanning tree because it contains a cycle 1–3–4–1. We refer to those arcs contained in a given spanning tree as *tree arcs* and to those arcs not contained in a given spanning tree as *nontree arcs*. The following two elementary observations will arise frequently in our development in this chapter.

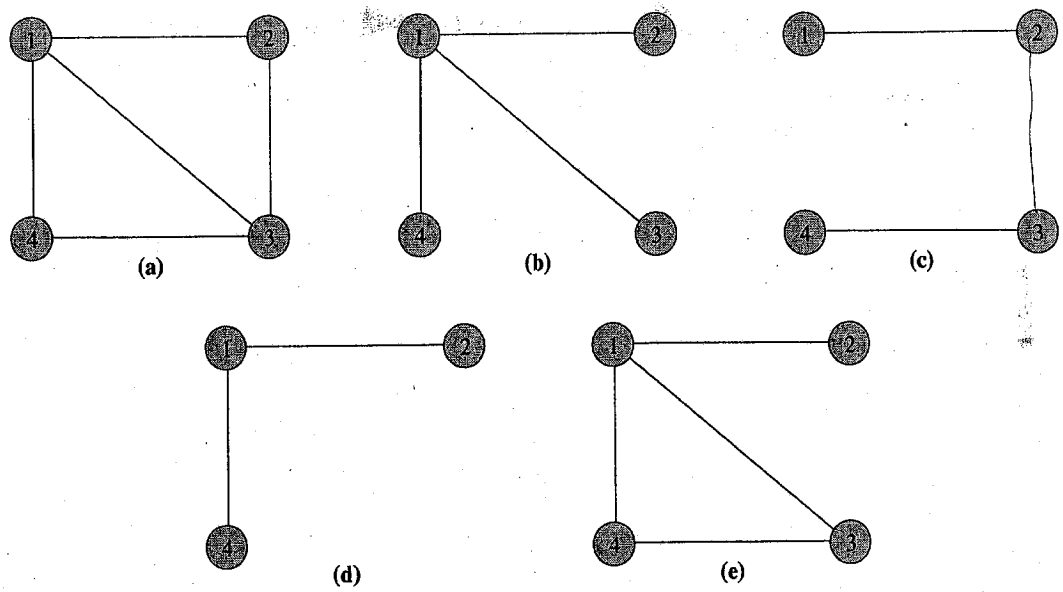


Figure 13.3 Illustrating spanning trees: (a) underlying graph; (b) two spanning trees; (c) nonspanning tree (disconnected graph); (d) nonspanning tree (doesn't span all nodes); (e) another nonspanning tree (cyclic graph).

1. For every nontree arc (k, l), the spanning tree T contains a unique path from node k to node l. The arc (k, l) together with this unique path defines a cycle [see Figure 13.4(a)].
2. If we delete any tree arc (i, j) from a spanning tree, the resulting graph partitions the node set N into two subsets [see Figure 13.4(b)]. The arcs from the un-

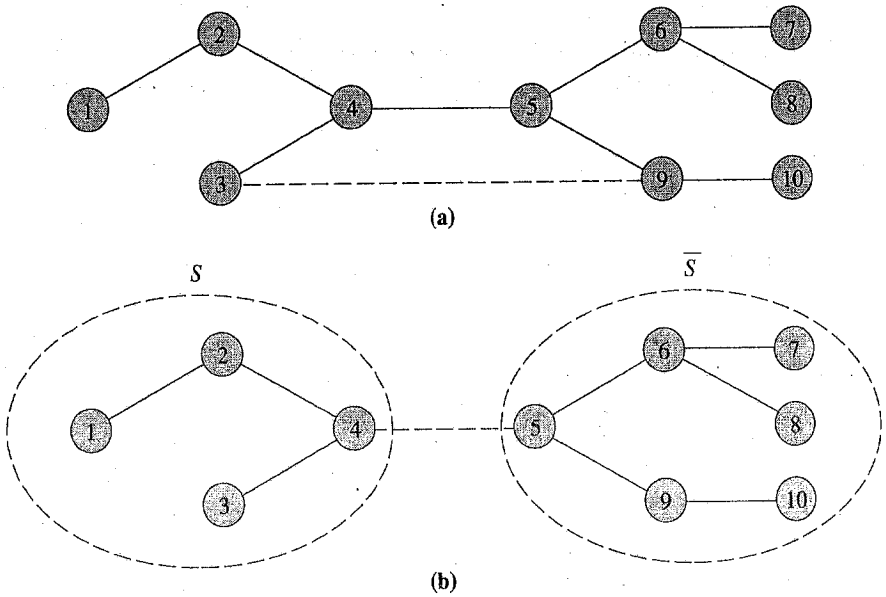


Figure 13.4 Illustrating properties of a spanning tree: (a) adding arc (3, 9) to the spanning tree forms the unique cycle 3–4–5–9–3; (b) deleting arc (4, 5) forms the cut $[S, \overline{S}]$ with $S = \{1, 2, 3, 4\}$.

derlying graph G whose two endpoints belong to the different subsets constitute a cut.

We next prove the two optimality conditions.

Theorem 13.1 (Cut Optimality Conditions). *A spanning tree T^* is a minimum spanning tree if and only if it satisfies the following cut optimality conditions: For every tree arc $(i, j) \in T^*$, $c_{ij} \leq c_{kl}$ for every arc (k, l) contained in the cut formed by deleting arc (i, j) from T^*.*

Proof. It is easy to see that every minimum spanning tree T^* must satisfy the cut optimality condition. For, if $c_{ij} > c_{kl}$ and arc (k, l) is contained in the cut formed by deleting arc (i, j) from T^*, then introducing arc (k, l) into T^* in place of arc (i, j) would create a spanning tree with a cost less than T^*, contradicting the optimality of T^*.

We next show that if any tree T^* satisfies the cut optimality conditions, it must be optimal. Suppose that T° is a minimum spanning tree and $T^\circ \neq T^*$. Then T^* contains an arc (i, j) that is not in T° (the reader might find it helpful to refer to Figure 13.5 while reading the rest of the proof). Deleting arc (i, j) from T^* creates a cut, say $[S, \bar{S}]$. Now notice that if we add the arc (i, j) to T°, we create a cycle W that must contain an arc (k, l) [other than arc (i, j)] with $k \in S$ and $l \in \bar{S}$. Since T^* satisfies the cut optimality conditions, $c_{ij} \leq c_{kl}$. Moreover, since T° is an optimal spanning tree, $c_{ij} \geq c_{kl}$, for otherwise we could improve on its cost by replacing arc (k, l) by arc (i, j). Therefore, $c_{ij} = c_{kl}$. Now if we introduce arc (k, l) in the tree T^* in place of arc (i, j), we produce another minimum spanning tree and it has one more arc in common with T°. Repeating this argument several times, we can transform T^* into the minimum spanning tree T°. This construction shows that T^* is also a minimum spanning tree and completes the proof of the theorem. ◆

The cut optimality conditions imply that every arc in a minimum spanning tree is a minimum cost arc across the cut that is defined by removing it from the tree. In fact, the cut optimality conditions also imply that we can always include *any* minimum cost arc in any cut in some minimum spanning tree, which we state in the following somewhat stronger form.

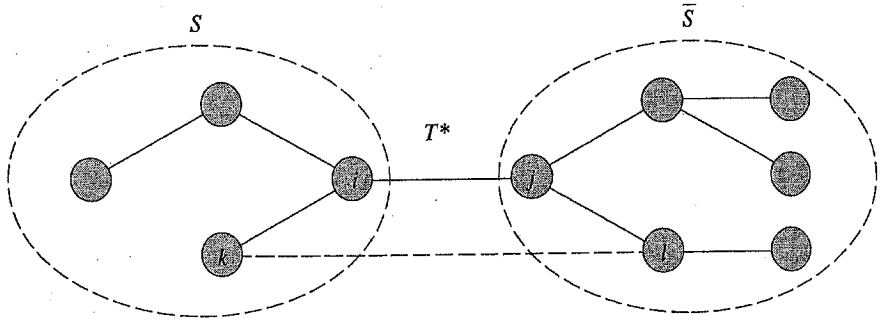


Figure 13.5 Proving cut and path optimality conditions.

Property 13.2. *Let F be a subset of arcs in some minimum cost spanning tree and let S be a set of nodes of some component of F. Suppose that (i, j) is a minimum cost arc in the cut $[S, \bar{S}]$. Then some minimum spanning tree contains all the arcs in F as well as the arc (i, j).*

Proof. Suppose that F is a subset of the minimum cost tree T^*. If $(i, j) \in T^*$, we have nothing to prove. So suppose that $(i, j) \notin T^*$. Adding (i, j) to T^* creates a cycle C, and C contains at least one arc $(p, q) \neq (i, j)$ in $[S, \bar{S}]$. By assumption, $c_{ij} \leq c_{pq}$. Also, since T^* satisfies the cut optimality conditions, $c_{ij} \geq c_{pq}$. Consequently, $c_{ij} = c_{pq}$, so adding arc (i, j) to T^* and removing arc (p, q) produces a minimum spanning tree containing all the arcs in F as well as the arc (i, j). ◆

The cut optimality conditions provide us with an "external" characterization of a minimum spanning tree that rests on the relationship between a single arc in the tree and many arcs outside the tree, that is, those in the cut that we produce by removing the arc from the tree. The following related path optimality conditions provide an alternative "internal" characterization that considers the relationship between a single nontree arc and several arcs in the tree, that is, those in the path formed by adding the nontree arc to the spanning tree.

Theorem 13.3 (Path Optimality Conditions). *A spanning tree T^* is a minimum spanning tree if and only if it satisfies the following path optimality conditions: For every nontree arc (k, l) of G, $c_{ij} \leq c_{kl}$ for every arc (i, j) contained in the path in T^* connecting nodes k and l.*

Proof. It is easy to show the necessity of the path optimality conditions. Suppose T^* is a minimal spanning tree satisfying these conditions and arc (i, j) is contained in the path in T^* connecting nodes k and l. If $c_{ij} > c_{kl}$, introducing arc (k, l) into T^* in place of arc (i, j) would create a spanning tree with a cost less than T^*, contradicting the optimality of T^*.

We establish the sufficiency of the path optimality conditions by using the sufficiency of the cut optimality conditions. This proof technique highlights the equivalence between these conditions. We will show that if a tree T^* satisfies the path optimality conditions, it must also satisfy the cut optimality conditions; Theorem 13.1 would then imply that T^* is an optimal tree. Let (i, j) be any tree arc in T^*, and let S and $\bar{S}$ be the two sets of connected nodes produced by deleting arc (i, j) from T^*. Suppose $i \in S$ and $j \in \bar{S}$. Consider any arc $(k, l) \in [S, \bar{S}]$ (see Figure 13.5). Since T^* contains a unique path joining nodes k and l and since arc (i, j) is the only arc in T^* joining a node in S and a node in $\bar{S}$, arc (i, j) must belong to this path. The path optimality condition implies that $c_{ij} \leq c_{kl}$; since this condition must be valid for every nontree arc (k, l) in the cut $[S, \bar{S}]$ formed by deleting any tree arc (i, j), T^* satisfies the cut optimality conditions and so it must be a minimum spanning tree. ◆

In the preceding discussion we have established two optimality conditions for the minimum spanning tree problem. The following optimality conditions for the maximum spanning tree problem are similar. We leave their proofs as an exercise (see Exercise 13.9).

Theorem 13.4 (Maximum Spanning Tree Optimality Conditions).

(*a*) *A spanning tree T^* is a maximum spanning tree if and only if it satisfies the following cut optimality conditions: For every tree arc $(i, j) \in T^*$, $c_{ij} \geq c_{kl}$ for every arc (k, l) contained in the cut formed by deleting arc (i, j) from T^*.*

(*b*) *A spanning tree is a maximum spanning tree T^* if and only if it satisfies the following path optimality conditions: For every nontree arc (k, l) of G, $c_{ij} \geq c_{kl}$ for every arc (i, j) contained in the tree path in T^* connecting nodes k and l.*

13.4 KRUSKAL'S ALGORITHM

The path optimality conditions immediately suggest the following straightforward algorithm for solving the minimum spanning tree problem. We start with any arbitrary spanning tree T and test the path optimality conditions. If T satisfies this condition, it is an optimal tree; otherwise, $c_{ij} > c_{kl}$ for some nontree arc (k, l) and some tree arc (i, j) contained in the unique path in T connecting nodes k and l. In this case, adding arc (k, l) to T in place of arc (i, j) gives us a spanning tree with a lower cost. Repeating this step will give us a minimum spanning tree within a finite number of iterations. Although this algorithm is strikingly simple, its running time cannot be polynomially bounded in the size of the problem data.

Simple Version of Kruskal's Algorithm

To derive an alternative and more efficient algorithm, known as *Kruskal's algorithm*, from the path optimality conditions, we consider an algorithm that builds an optimal spanning tree from scratch by adding one arc at a time. We first sort all the arcs in nondecreasing order of their costs and define a set, LIST, that is the set of arcs we have chosen as part of a minimum spanning tree. Initially, the set LIST is empty. We examine the arcs in the sorted order one by one and check whether adding the arc we are currently examining to LIST creates a cycle with the arcs already in LIST. If it does not, we add the arc to LIST; otherwise, we discard it. We terminate when $|\text{LIST}| = n - 1$. At termination, the arcs in LIST constitute a minimum spanning tree T^*.

The correctness of Kruskal's algorithm follows from the fact that we discarded each nontree arc (k, l) with respect to T^* at some stage because it created a cycle with the arcs already in LIST. But observe that the cost of arc (k, l) is greater than or equal to the cost of every arc in that cycle because we examined the arcs in the nondecreasing order of their costs. Therefore, the spanning tree T^* satisfies the path optimality conditions, so it is an optimal tree.

To illustrate Kruskal's algorithm on a numerical example, we consider the network shown in Figure 13.6(a). Sorted in the order of their costs, the arcs are (2, 4), (3, 5), (3, 4), (2, 3), (4, 5), (2, 1), and (3, 1). In the first three iterations, the algorithm adds the arcs (2, 4), (3, 5), and (3, 4) to LIST [see Figures 13.6(b) to (d)]. In the next two iterations, the algorithm examines arcs (2, 3) and (4, 5) and discards them because the addition of each arc to LIST creates a cycle [see Figure 13.6(e) and (f)]. Then the algorithm adds arc (2, 1) to LIST and terminates. Figure 13.6(g) shows the minimum spanning tree.

We might view the running time of Kruskal's algorithm as being composed of

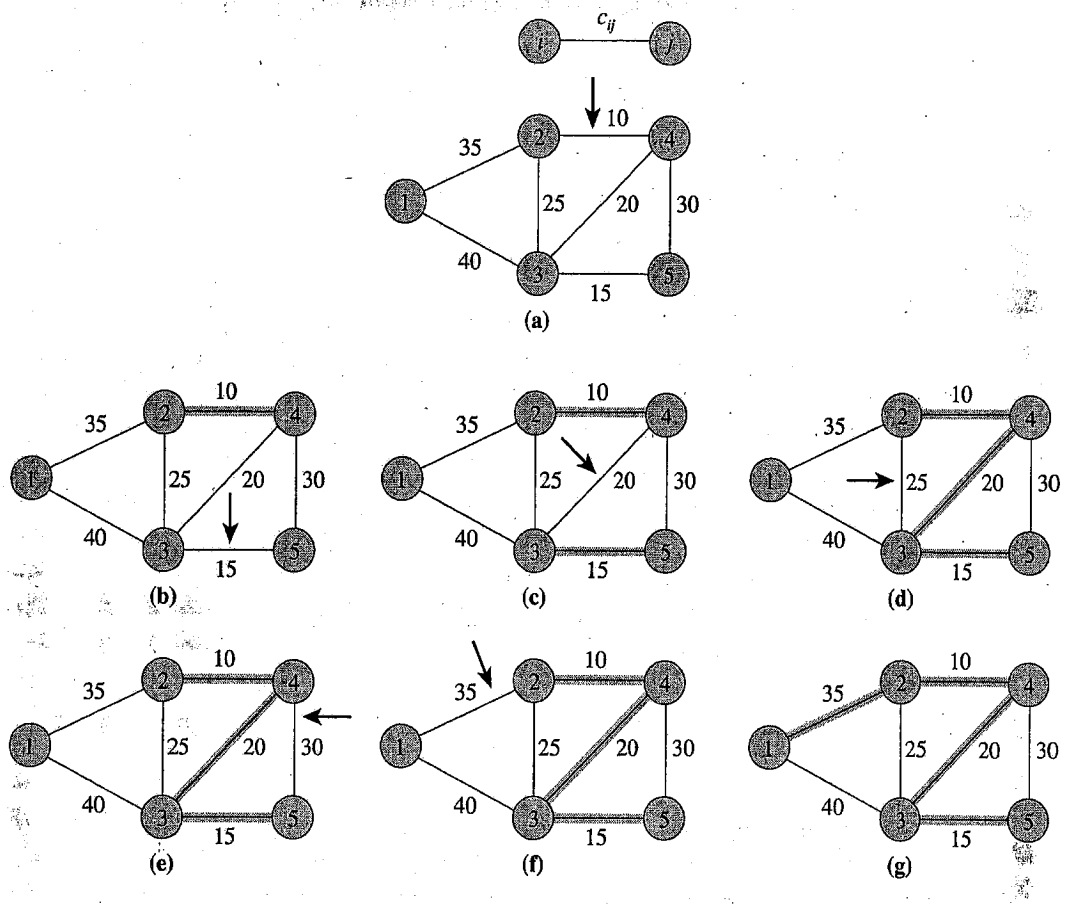


Figure 13.6 Illustrating Kruskal's algorithm.

the time for sorting the arcs and the time for detecting cycles. For a network with arbitrarily large arc costs, sorting requires $O(m \log m) = O(m \log n^2) = O(m \log n)$ time. The time to detect a cycle depends on the method we use for this step. One naive method would work as follows. The set LIST at any stage of the algorithm is a *forest* (i.e., a collection of subtrees). For example, the set LIST corresponding to Figure 13.6(c) consists of three trees containing the nodes {1}, {2, 4}, and {3, 5}, respectively. We denote these sets of nodes for a collection of trees by N_1, N_2, N_3, ... We can store these sets as different singly linked lists. While examining an arc (k, l), we scan through these linked lists and check whether both the nodes k and l belong to the same list. If so, adding arc (k, l) to LIST creates a cycle and we discard this arc. If nodes k and l belong to different lists, we add arc (k, l) to LIST, which requires merging the lists containing nodes k and l into a single list. Clearly, this data structure requires $O(n)$ time for each arc that we examine, so if we use this data structure, Kruskal's algorithm runs in $O(nm)$ time.

Improved Implementation of Kruskal's Algorithm

We now describe a more efficient implementation of Kruskal's algorithm that runs in $O(m + n \log n)$ time plus the time taken for sorting the arcs. This implementation is similar to the preceding one: We store the collection of trees, denoted by the sets

$N_1, N_2, N_3, \ldots$, as different singly linked lists. With each list L, we maintain two indices: *size(L)*, representing the number of elements in the list L; and *last(L)*, representing the last element in the list L. For each element i, we associate an index *first(i)* that stores the first element in the list containing node i. For example, if $L = \{1, 5, 6\}$, then size(L) = 3; last(L) = 6; and first(1) = first(5) = first(6) = 1.

Using this scheme, we can easily check whether the nodes k and l belong to the same list. If first(k) = first(l), they do; otherwise, they don't. This step contributes a total of $O(m)$ time to the running time of the algorithm. When we add the arc (k, l) to LIST, we need to merge the lists containing the nodes k and l. To merge these two lists, we always put the larger list first, breaking ties arbitrarily. The size indices allow us to determine the larger list, and last indices allow us to determine the last element of the larger list where we append the smaller list. As a result of the merge operation, several indices change. Updating the size and last indices is easy and requires $O(1)$ effort. To update the first indices, we need to modify this index for every node in the smaller list; the time required for this operation is proportional to the number of elements in the list. Suppose that the time required to merge the two lists L and L' of sizes h and h' (with $h \leq h'$) is ph for some constant p. We shall show that the total time required in all the mergings is $O(n \log n)$.

We prove this result using an induction argument. We claim that the total time required to obtain a merged list of size n is at most $pn \log n$ for some constant p. This result is clearly true for $n = 1$, since this case requires no mergings. Let us assume inductively that the result is true for any number of elements strictly less than n.

When we carry out the algorithm, we ultimately obtain a single list of n elements, obtained by appending two smaller lists L and L' containing h and $n - h$ elements. Let us assume that $h \leq n/2$, so that we place list L after list L' in the merge operation. By the inductive hypothesis, the time needed to create L is at most $ph \log h$ and the time needed to create L' is at most $p(n - h) \log(n - h)$. Since the time needed for merging the two lists is at most ph, the total time required for all the merging steps is at most

$$\begin{aligned} ph \log h + p(n - h) \log(n - h) + ph &\leq ph \log(n/2) + p(n - h)\log n + ph \\ &= ph(\log n - 1) + p(n - h)\log n + ph \\ &= pn \log n. \end{aligned}$$

The following theorem summarizes the implication of this result.

Theorem 13.5. *The improved implementation of Kruskal's algorithm solves the minimum spanning tree problem in $O(m + n \log n)$ time plus the time for sorting the arcs.*

Kruskal's algorithm requires two basic operations on lists of elements, which are commonly known as *union-find* operations. The *union* operation merges two lists and the *find* operation determines the list an element belongs to. Although our implementation of Kruskal's algorithm gives an attractive running time of $O(m + n \log n)$, in addition to the time for sorting, we could improve on this time even further by using better implementations of the union-find operations. The improved

implementation has a running time of $O(m\,\alpha(n, m))$ for a function $\alpha(n, m)$ that grows so slowly that for all practical purposes it can be viewed as a constant less than 6 (see the reference notes).

13.5 PRIM'S ALGORITHM

Just as the path optimality conditions allowed us to develop Kruskal's algorithm, the cut optimality conditions permit us to develop another simple algorithm for the minimum spanning tree problem, known as *Prim's algorithm*. This algorithm builds a spanning tree from scratch by fanning out from a single node and adding arcs one at a time. It maintains a tree spanning on a subset S of nodes and adds a nearest neighbor to S. The algorithm does so by identifying an arc (i, j) of minimum cost in the cut $[S, \bar{S}]$. It adds arc (i, j) to the tree, node j to S, and repeats this basic step until $S = N$. The correctness of the algorithm follows directly from Property 13.2 since this result implies that each arc that we add to the tree is contained in some minimum spanning tree with the arcs that we have selected in the previous steps.

We illustrate Prim's algorithm on the same example, shown in Figure 13.7(a), that we used earlier to illustrate Kruskal's algorithm. Suppose, initially, that $S = \{1\}$. The cut $[S, \bar{S}]$ contains two arcs, (1, 2) and (1, 3), and the algorithm selects the arc (1, 2) [see Figure 13.7(b)]. At this point $S = \{1, 2\}$ and the cut $[S, \bar{S}]$ contains the arcs (1, 3), (2, 3), and (2, 4). The algorithm selects arc (2, 4) since it has the minimum cost among these three arcs [see Figure 13.7(c)]. In the next two iterations, the algorithm adds arc (4, 3) and then arc (3, 5); Figure 13.7(d) and (e) show the details of these iterations. Figure 13.7(f) shows the minimum spanning tree produced by the algorithm.

To analyze the running time of Prim's algorithm, we consider each of the $n - 1$ iterations that the algorithm performs as it adds one arc at a time to the tree until it has a spanning tree with $n - 1$ arcs. In each iteration, the algorithm selects

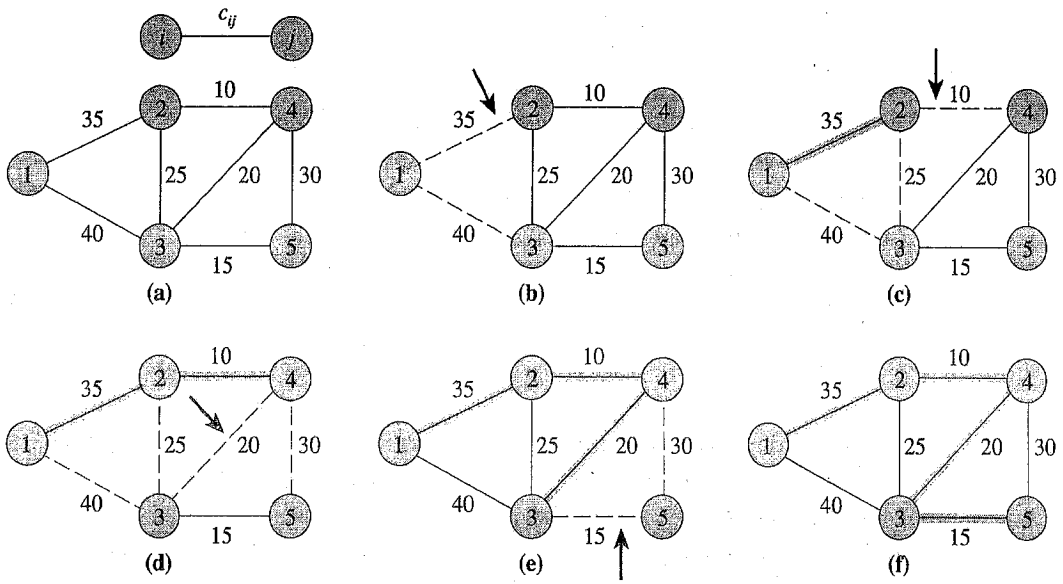


Figure 13.7 Illustrating Prim's algorithm.

the minimum cost arc in the cut $[S, \bar{S}]$. If we scan the entire arc list to identify the minimum cost arc, this operation requires $O(m)$ time, giving us an $O(nm)$ time bound for the algorithm. Therefore, this implementation of Prim's algorithm runs in $O(nm)$ time. However, we can improve upon it substantially, as we next show.

The bottleneck step in the $O(nm)$ implementation of Prim's algorithm is the identification of a minimum cost arc in the cut $[S, \bar{S}]$. We can improve the efficiency of this step by maintaining two indices for each node j in $\bar{S}$: (1) a distance label $d(j)$, which represents the minimum cost of arcs in the cut incident to a node j not in S (i.e., $d(j) = \min\{c_{ij}:(i, j) \in [S, \bar{S}]\}$), and (2) a predecessor label $pred(j)$, which represents the other endpoint of the minimum cost arc in the cut incident to node j. For example, in Figure 13.7(d), three arcs, (1, 3), (2, 3), and (4, 3), in the cut are incident to node 3. Among these arcs, arc (4, 3) has the minimum cost of 20. Therefore, $d(3) = 20$ and pred(3) = 4. For the same figure, $d(5) = 30$ and pred(5) = 4. If we maintain these indices, we can easily find the minimum cost of an arc in the cut; we simply compute $\min\{d(j):j \in \bar{S}\}$. If node i achieves this minimum, (pred(i), i) is a minimum cost arc in the cut. Observe that if we move node i from $\bar{S}$ to S, we need to update the distance and predecessor labels only for the nodes adjacent to node i.

Notice the similarity between this implementation of Prim's algorithm and the implementation of Dijkstra's algorithm that we discussed in Section 4.5. Just as in Dijkstra's algorithm, the basic operations are finding the minimum distance label $d(i)$ among the nodes in the set $\bar{S}$, moving the corresponding node into the set S, and updating the distance labels of those nodes in $\bar{S}$ that are adjacent to node i. Indeed, we can implement Prim's algorithm using the various types of heaps (or priority queues) that we used in our implementations of Dijkstra's algorithm in Section 4.7. Recall from Appendix A that a heap is a data structure that permits us to perform the following operations on a collection H of objects, each having an associated real number called its *key*.

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

find-min(H). Find and return an object from H with the minimum key.

insert(i, H). Insert a new object i with a predefined key into a collection H of objects.

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

delete-min(i, H). Delete an object i with the minimum key from the collection H of objects.

Observe that if we implement Prim's algorithm using a heap, H would be the collection of nodes in $\bar{S}$ and the key of a node would be its distance label. Prim's algorithm would be implemented as described in Figure 13.8. As always, we let C denote the maximum arc cost in the graph G.

As is clear from its description, Prim's algorithm performs the operations find-min, delete-min, and insert at most n times and the operation decrease-key at most

```
algorithm heap-Prim;
begin
    create-heap(H);
    for each j ∈ N − {1} do d(j) : = C + 1;
    set d(1) : = 0; and pred(1) : = 0;
    for each j ∈ N do insert(j, H);
    T* : = ∅;
    while |T*| < (n − 1) do
    begin
        find-min(i, H);
        delete-min(i, H);
        T* : = T* ∪ (pred(i), i);
        for each (i, j) ∈ A(i) with j ∈ H do
            if d(j) > c_ij then
            begin
                d(j) : = c_ij;
                pred(j) : = i;
                decrease-key(j, c_ij, H);
            end;
    end;
        T* is a minimum spanning tree;
end;
```

Figure 13.8 Prim's algorithm.

m times. When implemented with different heaps, the algorithm would have the running times shown in Figure 13.9.

Our discussion of the binary heap, d-heap, and Fibonacci heap data structures in Appendix A permits us to justify these time bounds. In that discussion we show that the d-heap data structure performs each delete-min operation in $O(d \log_d n)$ time and every other heap operation in $O(\log_d n)$ time. If we select $d = m/n$, this result gives us a running time of $O(m \log_d n + nd \log_d n) = O(m \log_d n)$ for Prim's algorithm implemented using the d-heap data structure. The binary heap is a special case of d-heap with $d = 2$, so its time bound is $O(m \log n)$. The Fibonacci heap data structure performs each delete-min operation in $O(\log n)$ time and every other heap operation in $O(1)$ time. Consequently, the Fibonacci heap implementation of Prim's algorithm runs in $O(m + n \log n)$ time.

Theorem 13.6. *Fibonacci heap implementation of Prim's algorithm solves the minimum spanning tree problem in $O(m + n \log n)$ time.* ◆

Heap type	Running time
Binary heap	$O(m \log n)$
d-heap	$O(m \log_d n)$, with $d = \max\{2, m/n\}$
Fibonacci heap	$O(m + n \log n)$
Johnson's data structure	$O(m \log \log C)$

Figure 13.9 Running times of various heap implementations of Prim's algorithm.

13.6 SOLLIN'S ALGORITHM

We can use the cut optimality conditions to derive another novel algorithm for the minimum spanning tree problem, known as Sollin's algorithm. We can view this algorithm as a hybrid version of Kruskal's and Prim's algorithm. As in Kruskal's algorithm, Sollin's algorithm maintains a collection of trees spanning the nodes N_1, N_2, N_3, . . . , and adds arcs to this collection. However, at every iteration, it adds minimum cost arcs emanating from these trees, an idea borrowed from Prim's algorithm. As a result, we obtain a fairly simple algorithm that uses elementary data structures and runs in $O(m \log n)$ time. As pointed out in the reference notes, a more clever implementation of this approach runs in $O(m \log \log n)$ time.

Sollin's algorithm repeatedly performs the following two basic operations:

nearest-neighbor (N_k, i_k, j_k). This operation takes as an input a tree spanning the nodes N_k and determines an arc (i_k, j_k) with the minimum cost among all arcs emanating from N_k [i.e., $c_{i_k j_k} = \min\{c_{ij}: (i, j) \in A, i \in N_k \text{ and } j \notin N_k\}$]. To perform this operation we need to scan all the arcs in the adjacency lists of nodes in N_k, and find a minimum cost arc among those arcs that have one endpoint not belonging to N_k.

merge(i_k, j_k). This operation takes as an input two nodes i_k and j_k, and if the two nodes belong to two different trees, then merges these two trees into a single tree.

Using these two basic operations, we state Sollin's algorithm as shown in Figure 13.10.

We illustrate Sollin's algorithm on the same numerical example that we have used to illustrate Kruskal's and Prim's algorithms. As shown in Figure 13.11(b), Sollin's algorithm starts with a forest containing five trees: Each tree is a singleton node. This figure also shows the least cost arc emanating from each tree. We next perform mergings, reducing the number of trees to only two [see Figure 13.11(c)]. The least cost arc emanating from these two trees is (3, 4), and when we add this arc, we obtain the spanning tree shown in Figure 13.11(d). The algorithm now terminates.

To analyze the running time of Sollin's algorithm, we need to discuss the data structure needed to implement it. We will show that the algorithm performs $O(\log n)$ executions of the while loop, and that we can perform all the nearest-neighbor and

```
algorithm Sollin;
begin
    for each i ∈ N do N_i := {i};
    T* := ∅;
    while |T*| < (n − 1) do
    begin
        for each tree N_k do nearest-neighbor(N_k, i_k, j_k);
        for each tree N_k do
            if nodes i_k and j_k belong to different trees then
                merge(i_k, j_k) and update T* := T* ∪ {(i_k, j_k)};
    end;
end;
```

Figure 13.10 Sollin's algorithm.

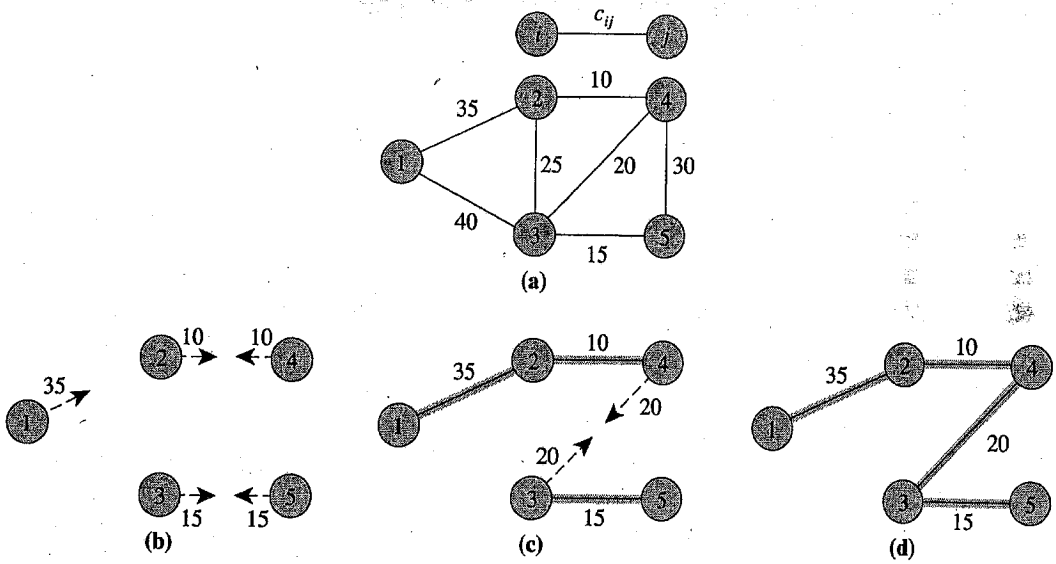


Figure 13.11 Illustrating Sollin's algorithm.

merge operations in $O(m)$ time. These results establish a time bound of $O(m \log n)$ for Sollin's algorithm.

We store the nodes of a tree as a circular doubly linked list. The doubly linked list allows us to visit every node of the tree starting at any tree node. We assign a numerical label with every node in the network; the label satisfies the following two properties: (1) nodes of the same tree have the same label, and (2) nodes of different trees have different labels. At the beginning of the algorithm, we assign label i to each node $i \in N$.

Using this data structure, we can easily check whether an arc (i, j) has both of its endpoints in the same tree: We answer this question simply by checking the labels of nodes i and j. This observation implies that we can perform the nearest-neighbor operation for each tree in the forest in a total time of $O(\sum_{i \in N} | A(i) |) = O(m)$.

We perform merge operations in a while loop using the following iterative scheme. In each iteration we select an unexamined tree, say N_1, and consider the minimum cost arc (i_1, j_1) emanating from N_1. (Node i_1 is in N_1 and j_1 might or might not be in N_1.) Suppose that nodes in N_1 have the label α. If node j_1 also has the label α, the iteration ends. Otherwise, we scan through the nodes of the tree, say N_2, containing node j_1 and assign them the label α. Next, we consider the minimum cost arc (i_2, j_2) emanating from N_2. If node j_2 has label α, the iteration ends; otherwise, we scan through the nodes of the tree, say N_3, containing node j_2 and assign them the label α. We repeat this process until the iteration ends. Notice that within an iteration we might assign the nodes of several trees the label of the first tree. When an iteration ends, we initiate a new iteration by selecting another unexamined tree. We terminate this iterative process when we have examined all the trees. As is clear from this description, this method assigns a label to each node once and hence runs in $O(n)$ time.

Having proved that each execution of the while loop in Sollin's algorithm re-

quires $O(m)$ time, we now obtain a bound on the number of executions of the loop. Each execution of the loop reduces the number of trees in the forest by a factor of at least two because we merge each tree into a larger tree. This observation implies that we will perform $O(\log n)$ executions of the loop. We have therefore established the following result.

Theorem 13.7. *The execution of Sollin's algorithm requires $O(m \log n)$ time.*

13.7 MINIMUM SPANNING TREES AND MATROIDS

In keeping with the orientation of this book, we have examined the minimum spanning tree problem from a perspective of graph theory and the data structures needed to implement spanning tree algorithms efficiently. We could, instead, view the minimum spanning tree problem and develop several of the core ideas of this chapter from at least two other perspectives: (1) broader notions in combinatorial optimization, and (2) linear programming. These two alternative viewpoints are instructive because they help to show the connection between network optimization and other important topics in discrete optimization. Indeed, the minimum spanning tree problem and network flows have inspired the development of many other problem domains in discrete optimization. Consequently, it is useful to pause at this point and briefly delineate these connections.

Matroids and the Greedy Algorithm

Suppose that we view a spanning tree in the following way: We have a finite collection of objects E, the arcs of a network, and we define a subset I of objects to be *independent* if they do not form a cycle in the network. If each object (i.e., arc) e has an associated weight w_e, the minimal spanning tree problem seeks an $n - 1$ element independent set I with the smallest total weight $w(I) \equiv \sum_{e \in I} w_{ij}$.

Let us now describe this and related problems in a more abstract setting. A *subset system* $(E, \mathcal{I})$ is a finite set of objects E and nonempty collection $\mathcal{I}$ of subsets of these objects, called *independent sets*, that satisfies the hereditary property that whenever I is an independent set (i.e., belongs to $\mathcal{I}$) and I' is a subset of I, then I' also is an independent set.

Suppose that we associate a weight w_e with each element e of E and define the weight $w(S)$ of any subset S of E as the sum of the weights of its elements; that is, $w(S) \equiv \sum_{e \in S} w_e$. As a generalization of the minimum spanning tree problem, we might consider the following *independence (or subset) system optimization problem*: Find a maximal independent set of the subset system $(E, \mathcal{I})$ with the minimum weight.

At this level of generality, the independence system optimization problem appears to be hopelessly difficult to solve efficiently. Therefore, we need to impose additional structure on the problem so that it becomes tractable. We would like to do so, however, by imposing the least amount of additional structure so that the results remain as general as possible. The following type of auxiliary structure appears to be just right for this purpose.

A subset system $(E, \mathcal{I})$ is a *matroid* if it satisfies the growth property that if I_p and I_{p+1} are independent sets containing p and $p + 1$ elements, we always can

find an element $e \in I_{p+1} - I_p$, satisfying the property that $I_p \cup \{e\}$ is an independent set.

Note that this definition implies if I and I' are any two independent sets and $|I'| > |I|$, we can add certain elements of I' to I and obtain another independent set I'' so that I'' contains I and has as many elements as I'. That is, the sets I_p and I_{p+1} in the definition need not differ in cardinality by one. (We establish this *extended growth property* by applying the growth property to I and the first $|I| + 1$ elements of I', which are independent by the definition, and then repeating the operation.)

Let us illustrate the definition of a matroid with a few examples.

Graphic or forest matroid. Note that forests in a network satisfy these definitions if we let E equal the arcs in a network and let $\mathcal{I}$ denote the collection of arc sets that contain no cycles (i.e., the arcs define a forest). In this case the system $(E, \mathcal{I})$ is an independent system because removing arcs from a forest always produces another forest. Moreover, if I_p and I_{p+1} are two forests containing p and $p + 1$ arcs, the forest I_{p+1} must contain an arc e that we can add to I_p and produce another forest $I_p \cup \{e\}$ (see Exercise 13.41). Consequently, the system $(E, \mathcal{I})$ is a matroid.

Partition matroid. Let $E = E_1 \cup E_2 \cup \cdots \cup E_K$ be a union of K disjoint finite sets and let $u_1, u_2, \ldots, u_K$ be given positive integers. Let $\mathcal{I}$ be the family of subsets I of E that satisfy the property that for all $k = 1, 2, \ldots, K$, I contains no more than u_k elements of E_k. The system is a matroid. Note that if we consider a bipartite graph $(N_1 \cup N_2, A)$ and let E_k, for all nodes k in N_1, be the set of arcs incident to node k, then if all the u_k are equal to 1, the matroid defines "half" of an assignment problem. Another partition matroid defined on the nodes N_2 defines the other half of the matroid, so any feasible solution to the assignment problem is an independent set in both partition matroids.

Matric matroid. Let M be a real-valued matrix, let E be the columns of M, and let $\mathcal{I}$ be sets of columns of M that are linearly independent. Since removing columns from a linearly independent set of columns produces another independent set, the system $(E, \mathcal{I})$ is a subset system. By elementary results in linear algebra, this system also satisfies the growth property and so is a matroid.

Let us make one further observation about matroids. A *maximal independent set* is an independent set I satisfying the property that we cannot add any other element e to I and produce another independent set. The (extended) growth property implies that every maximal independent set of a matroid contains the same number of elements (since we can always add elements of one maximal independent set to another if they contain a different number of elements). Borrowing notation from linear algebra, we refer to any maximal independent set as a *basis* of the matroid. In this terminology, the matroid optimization problem seeks a basis with the smallest possible total weight.

We can attempt to solve this problem by using a *greedy algorithm* (Figure 13.12) which is a direct generalization of Kruskal's algorithm.

Note that for the minimal spanning tree problem, the test condition "LIST $\cup$ $\{e_j\}$ is independent" from this algorithm is just the test condition from Kruskal's

```
algorithm greedy;
begin
    order the elements of E = {e1, e2, . . . , eK} so that w1 ≤ w2 ≤ ··· ≤ wK;
    set LIST : = ∅;
    for j = 1 to K do
        if LIST ∪ {ej} is independent then LIST : = LIST ∪ {ej};
    LIST is a minimum weight basis;
end;
```

Figure 13.12 Greedy algorithm.

algorithm, namely, that the network defined by the arcs in LIST and e_j contains no cycle.

Theorem 13.8. *The greedy algorithm solves the matroid optimization problem.*

Proof. Let I^* be any optimal solution to the matroid optimization problem and let LIST $= \{e_{j_1}, e_{j_2}, \ldots, e_{j_n}\}$ be the solution generated by the greedy algorithm. We will show that $w(\text{LIST}) = w(I^*)$ and therefore that LIST is an optimal basis as well. If LIST $= I^*$, we have nothing to prove. So assume that LIST $\neq I^*$. Suppose that we order the elements of I^* in the order of increasing indices from the set $E = \{e_1, e_2, \ldots, e_K\}$ as $e_{j_1}, e_{j_2}, \ldots, e_{j_k}, e_q, \ldots$, with $e_q \neq e_{j_{k+1}}$ and assume that e_q is the first element of I^* not in LIST. Since the set $\{e_{j_1}, e_{j_2}, \ldots, e_{j_k}, e_q\}$ is independent, the steps of the greedy algorithm imply that $q \geq j_{k+1}$ and therefore that $w_q \geq w_{j_{k+1}}$. Since both the sets $I = \{e_{j_1}, e_{j_2}, \ldots, e_{j_k}, e_{j_{k+1}}\}$ and I^* are independent, the growth property implies that we can add elements of I^* to I to obtain another basis I'. Since this basis contains the elements $I^* \cup \{e_{j_{k+1}}\} - e_p$ for some $e_p \in I^*$ and $p \geq j_{k+1}$, $w(I') \leq w(I^*)$; consequently, I' is also an optimal basis. Note that this basis has a greater number of lead elements in common with LIST (at least $k + 1$). But now if we apply the same argument to the sets I' and LIST, we will obtain another optimal basis with at least one more lead element in common with LIST. If we continue in this fashion, eventually I' will equal LIST, therefore establishing that LIST is a minimum-basis matroid. ◆

Note that this discussion not only gives an alternative proof of Kruskal's algorithm for the minimum spanning tree problem, but also shows that two underlying combinatorial properties—independence and the growth property—are the essential ingredients necessary to ensure that the greedy algorithm solves the minimum spanning tree problem. (In Exercise 13.45 we show that the greedy algorithm will solve the minimum weight independent set problem for any choice of the element weights if and only if the subset system is a matroid.) Therefore, any other property of a graph is irrelevant for ensuring that Kruskal's algorithm works correctly. That is, we have now identified the combinatorial postulates that drive the algorithm.

13.8 MINIMUM SPANNING TREES AND LINEAR PROGRAMMING

Linear programming provides yet another proof of Kruskal's algorithm. Moreover, the development of a linear programming-based approach permits us to make some elementary connections between network optimization and an important topic in

applied mathematics, polyhedral combinatorics, which is the study of integer polyhedra (i.e., polyhedra with integer extreme points). As shown in Section 11.12, the minimum cost flow problem provides another connection between these topics.

Let $A(S)$ denote the set of arcs contained in the subgraph of $G = (N, A)$ induced by the node set S [i.e., $A(S)$ is the set of arcs of A with both endpoints in S]. Consider the following integer programming formulation of the minimum spanning tree problem:

$$\text{Minimize} \quad \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{13.2a}$$

subject to

$$\sum_{(i,j)\in A} x_{ij} = n - 1, \tag{13.2b}$$

$$\sum_{(i,j)\in A(S)} x_{ij} \leq |S| - 1 \qquad \text{for any set } S \text{ of nodes,} \tag{13.2c}$$

$$x_{ij} \geq 0 \text{ and integer.} \tag{13.2d}$$

In this formulation, the 0–1 variable x_{ij} indicates whether we select arc (i, j) as part of the chosen spanning tree (note that the second set of constraints with $|S| = 2$ implies that each $x_{ij} \leq 1$). The constraint (13.2b) is a cardinality constraint implying that we choose exactly $n - 1$ arcs, and the "packing" constraint (13.2c) implies that the set of chosen arcs contain no cycles (if the chosen solution contained a cycle, and S were the set of nodes on a chosen cycle, the solution would violate this constraint). Note that as a function of the number of nodes in the network, this model contains an exponential number of constraints. Nevertheless, as we will show, we can solve it very efficiently by applying Kruskal's algorithm. We might note that any formulation in the variables x_{ij} always requires an exponential number of constraints; that is, we cannot replace the given constraints by some polynomial set of constraints and still have a valid formulation of the problem. Nevertheless, it is possible to give a polynomial formulation of the problem if we introduce new (multicommodity flow) variables (see the reference notes).

Suppose that we consider the linear programming relaxation of this integer programming model. That is, we drop the restriction that the variables be integer. As we noted in Section 9.4 (also see Appendix C), we can formulate a set of reduced cost and complementary slackness optimality conditions for every linear programming problem and use these conditions, as we use the reduced costs and complementary slackness conditions of network flows, to assess when a given feasible solution is optimal. Recall that for network flow problems, we used node potentials to define the reduced costs and the complementary slackness conditions; each node in a minimum cost flow problem corresponds to one equation of the mass balance constraints $\mathcal{N}x = b$, so we can view the potentials as associated with these equations. Since the minimum spanning tree formulation has one equation or inequality for any set S of nodes (the one equation in the model corresponds to the node set $S = N$), for the minimum spanning tree problem we associate a potential μ_S with every set S of nodes. The potential μ_N is unrestricted in sign and the other potentials μ_S must be nonnegative. We then define the reduced cost c^{μ}_{ij} of any arc as

$$c^{\mu}_{ij} = c_{ij} + \sum_{A(S) \text{ contains arc } (i,j)} \mu_S.$$

With this definition of the reduced costs, we have the following complementary slackness optimality conditions.

Minimum spanning tree complementary slackness optimality conditions. *A solution x of the minimum spanning tree problem is an optimal solution to the linear programming relaxation of the integer programming formulation (13.2) if and only if we can find node potentials μ_S defined on node sets S so that the reduced costs satisfy the following conditions:*

$$c_{ij}^{\mu} = 0 \qquad \text{if } x_{ij} > 0.$$

$$c_{ij}^{\mu} \geq 0 \qquad \text{if } x_{ij} = 0.$$

We can use this fundamental result to give yet another proof that Kruskal's algorithm solves the minimum spanning tree problem.

Theorem 13.9. *If x is the solution generated by Kruskal's algorithm, x solves both the integer program (13.2) and its linear programming relaxation.*

Rather than giving a formal proof of this theorem, let us illustrate the proof technique on the five-node example that we have already considered in Figure 13.6. For this problem, Kruskal's algorithm chooses the arcs of the minimum spanning tree in the order (2, 4), (3, 5), (3, 4), and (1, 2). So we set $x_{24} = x_{35} = x_{34} = x_{12} = 1$ and $x_{13} = x_{23} = x_{45} = 0$. Note that during the course of applying Kruskal's algorithm, we form several connected node components: first $\{2, 4\}$, then $\{3, 5\}$, then $\{2, 3, 4, 5\}$, and finally, the entire node set $\{1, 2, 3, 4, 5\}$. We will associate a nonzero potential with these sets and a zero potential with every other set of nodes. We define these potentials in the reverse order that Kruskal's algorithm formed the node components. We first set $\mu_{\{1,2,3,4,5\}} = -35$, the negative of the cost of the final arc added to the tree. Now we note that each arc (i, j) that we add to the tree defines a node component $S(i, j)$. For example, $S(3, 4) = \{2, 3, 4, 5\}$. Moreover, at some later stage in the algorithm, we combine the node component $S(i, j)$ with one or more other nodes to define a large component by adding another arc (p, q) to the tree. We now set the potential of the node component $S(i, j)$ to be the difference between the cost of arc (p, q) and the cost of arc (i, j). Therefore, we set $\mu_{\{2,3,4,5\}} = c_{12} - c_{34} = 35 - 20 = 15$, $\mu_{\{3,5\}} = c_{34} - c_{35} = 20 - 15 = 5$, and $\mu_{\{2,4\}} = c_{34} - c_{24} = 20 - 10 = 10$.

Now checking the reduced cost of every arc, we find that

$$c_{12}^{\mu} = 35 - 35 = 0$$

$$c_{13}^{\mu} = 40 - 35 = 5$$

$$c_{23}^{\mu} = 25 - 35 + 15 = 5$$

$$c_{24}^{\mu} = 10 - 35 + 15 + 10 = 0$$

$$c_{34}^{\mu} = 20 - 35 + 15 = 0$$

$$c_{35}^{\mu} = 15 - 35 + 15 + 5 = 0$$

$$c_{45}^{\mu} = 30 - 35 + 15 = 10.$$

Note that with these choices of the potentials, the reduced cost of every arc chosen by Kruskal's algorithm is zero and the cost of every other arc (i, j) is the difference between the cost of arc (i, j) and the cost of the most expensive arc on the path formed by adding arc (i, j) to the tree found by Kruskal's algorithm. It is fairly easy to use an induction argument to extend this proof technique for any problem and thus to give a formal proof of Theorem 13.9 (see Exercise 13.42).

The proof technique we have just illustrated establishes one of the most important core results in combinatorial optimization. Since a linear program always has an extreme point solution (see Appendix C for this result and for linear programming definitions), if we can show that for every choice of the coefficients of its objective function, a linear programming formulation has at least one integral solution, then the extreme points of the polyhedron defined by that linear program are integer valued. Since we have just established this property for the linear programming relaxation of the integer program (13.2), we have proven the following fundamental result.

Theorem 13.10. *The polyhedron defined by the linear programming relaxation of the packing formulation of the minimum spanning tree problem has integer extreme points.*

This theorem is just one example of an important meta rule that seems to lie at the core of combinatorial optimization; namely, for essentially most optimization problems that can be solved in polynomial time, it is possible to define a linear program with integer extreme points that contains the incident vectors of the solution to the combinatorial optimization problem. The minimum cost flow problem and the minimal spanning tree problem were two of the first notable examples of this result discovered in the combinatorial optimization literature; these results have inspired many streams of investigation within discrete optimization, such as the study of matroids that we introduced in Section 13.7. For example, it is possible to specify a linear programming formulation of the matroid optimization problem so that the extreme points of the linear programming formulation are exactly the set of bases of the underlying matroid (see Exercise 13.44).

13.9 SUMMARY

The minimum spanning tree problem is perhaps the simplest, and certainly one of the most central, models in the field of combinatorial optimization. In this chapter, after describing several applications of minimum spanning trees, we proved two (equivalent) necessary and sufficient conditions—the *cut and path optimality conditions*—for characterizing the optimality of minimum spanning trees. The cut optimality conditions state that a spanning tree T^* is a minimum spanning tree if and only if the cost of the tree arc (i, j) is less than or equal to the cost of every nontree arc in the cut formed by deleting arc (i, j) from T^*. The path optimality conditions are closely related to these conditions (in a sense, they are dual conditions); they state that a spanning tree T^* is a minimum spanning tree if and only if the cost of every nontree arc (k, l) is greater than or equal to the cost of every tree arc in the path in T^* between nodes k and l.

In this chapter we described three algorithms for solving the minimum spanning tree problem: Kruskal's, Prim's, and Sollin's. All these algorithms are easy to implement, have excellent running times, and are very efficient in practice. Figure 13.13 summarizes the basic features of these algorithms.

The minimum spanning tree problem is important not only because it is a core model in network optimization, but also because it serves as a valuable prototype model in combinatorial optimization that has stimulated many lines of inquiry. In this chapter we have considered two ways in which minimum spanning trees relate to general issues in combinatorial optimization. If we consider Kruskal's algorithm as a greedy procedure that chooses the minimum cost feasible arc at each step, we might ask whether a similar type of greedy algorithm is able to solve other combinatorial optimization problems. We have answered this question affirmatively by showing that the greedy algorithm also solves a broad class of problems known as matroid optimization problems.

Studying specialized structures, such as matroids, is one very important stream of inquiry in combinatorial optimization. Another is the use of linear programming as a tool for understanding and solving combinatorial optimization problems. In Section 13.8 we showed how to characterize the incidence vectors of spanning trees as solutions to a linear programming formulation of the problem; we also showed how to interpret Kruskal's algorithm as a method for solving this linear program. This development illustrates the use of linear programming in combinatorial optimization and is indicative of the type of investigations that analysts conduct in the important subspecialty of combinatorial optimization known as polyhedral combinatorics (i.e., the study of integer polyhedra).

Algorithm	Running time	Features
Kruskal's algorithm	$O(m + n \log n)$ plus time needed to sort m arc lengths	1. Examines arcs in nondecreasing order of their lengths and include them in the minimum spanning tree if the added arc does not form a cycle with the arcs already chosen. 2. The proof of the algorithm uses the path optimality conditions. 3. Attractive algorithm if the arcs are already sorted in increasing order of their lengths.
Prim's algorithm	$O(m + n \log n)$	1. Maintains a tree spanning a subset S of nodes and adds a minimum cost arc in the cut $[S, \bar{S}]$. 2. The proof of the algorithm uses the cut optimality conditions. 3. Can be implemented using a variety of heaps structures; the stated time bound is for the Fibonacci heap data structure.
Sollin's algorithm	$O(m \log n)$	1. Maintains a collection of node-disjoint trees: in each iteration, adds the minimum cost arc emanating from each such tree. 2. The proof of the algorithm uses the cut optimality conditions.

Figure 13.13 Summary of minimum spanning tree algorithms.

REFERENCE NOTES

Algorithms for the minimum spanning tree problem, developed as early as 1926, are among the earliest network algorithms. The paper by Graham and Hell [1985] presents an excellent survey of the historical developments of minimum spanning tree algorithms. Borůvka [1926] and Jarník [1930] independently formulated and solved the minimum spanning tree problem. Later, other researchers rediscovered these algorithms. Kruskal [1956] and Loberman and Weinberger [1957] independently discovered Kruskal's algorithm discussed in Section 13.4. Prim [1957] developed the algorithm described in Section 13.5. Sollin presented his algorithm, discussed in Section 13.6, in a seminar in 1961; it was never published. Claude Berge was present at this seminar and reported this algorithm in his book, Berge and Ghouila-Houri [1962]. Later, researchers discovered that Sollin's algorithm is similar to Borůvka's algorithm and that Prim's algorithm is similar to Jarník's algorithm.

Our description of Kruskal's algorithm runs in the time required to sort m numbers plus $O(m + n \log n)$. The use of improved *union-find* data structures leads to a faster implementation of Kruskal's algorithm. This implementation, as developed by Tarjan [1984], runs in the time required to sort m numbers plus $O(m\ \alpha(n, m))$; $\alpha(n, m)$ is the Ackermann function which, for all practical purposes, is smaller than 6. In this chapter we reported an $O(m + n \log n)$ implementation of Prim's algorithm; this implementation appears to be new. Gabow, Galil, Spencer, and Tarjan [1986] presented a variant of this algorithm that runs in $O(m \log \beta(m, n))$ time with the function $\beta(m, n)$ defined as $\beta(m, n) = \min\{i : \log^{(i)}(m/n) \leq 1\}$. In this expression, $\log^{(i)} x = \log \log \log \cdots \log x$ with the log iterated i times. So $\beta(m, n)$ is a very slowly growing function. For example, if $m/n = 2^{2^{64,000}}$ then $\beta(m, n) = 6$. Yao [1975] developed an improved implementation of Sollin's algorithm running in $O(m \log \log n)$ time. Currently, the fastest algorithm for solving the minimum spanning tree algorithm is Tarjan's [1984] implementation of Kruskal's algorithm if the arcs are already sorted, and Gabow et al. [1986] variant of Prim's algorithm, otherwise. Gabow et al. [1986] also give efficient algorithms that (1) solve the minimum spanning tree problem in a directed network (i.e., the arborescence problem), and (2) solve the minimum spanning tree problem with a single degree constraint.

Chin and Houch [1978], Gavish and Srikanth [1979], and Tarjan [1982] have developed techniques for reoptimizing the minimum spanning tree problem when we change arc costs. Haymond, Jarvis, and Shier [1980] have described data structures for implementing Kruskal's, Prim's, and Sollin's algorithm and have presented computational results for these algorithms. Jarvis and Whited [1983] described the results of another computational study. These studies indicate that Prim's and Sollin's algorithms are consistently superior to Kruskal's algorithm. They show that Sollin's algorithm is better than Prim's algorithm for sparse networks, and is worse for dense networks. These studies find that the best implementation of Prim's algorithm uses a variant of Dial's implementation of Dijkstra's algorithm that we described in Section 4.6.

Our presentation of matroids in Section 13.7 and of a linear programming formulation of the minimum spanning tree in Section 13.8 merely touches upon two very important topics in combinatorial optimization. Although the concept of matroids is quite old, dating from their introduction by Whitney [1935], their use in

combinatorial optimization is much more recent, stemming from the seminal contributions of Edmonds [1965c, 1971]. The book by Lawler [1976] and the survey paper by Bixby [1982] highlight connections between matroids and network optimization. The books by Tutte [1971] and Welsh [1976] provide excellent mathematical accounts of this field, and the book by Recski [1988] presents many illuminating applications in engineering and the physical sciences.

The description of the polyhedral structure of combinatorial optimization problems via linear programs has become a very fertile field in combinatorial optimization that has shed theoretical light on many problems and led to effective algorithms for solving many important applications. The comprehensive text by Nemhauser and Wolsey [1988] gives an instructive account of this field, known as *polyhedral combinatorics*. The linear programming description of the minimum spanning tree problem, and the interpretation of Kruskal's algorithm as a method for solving the linear programming formulation of the problem, has served as an important stimulus for developments of this field. As but one example, this approach has proven very fruitful in developing algorithms for solving the nonbipartite matching problems that we considered in Chapter 12 from a purely combinatorial approach. For a polynomial formulation of the minimal spanning tree problem using multicommodity flow variables, see the survey by Magnanti, Wolsey, and Wong [1992].

The applications of the minimum spanning tree problem that we presented in Section 13.2 are adapted from the following papers:

1. Designing physical systems (Borůvka [1926], Prim [1957], Loberman and Weinberger [1957], and Dijkstra [1959])
2. Optimal message passing (Prim [1957])
3. All pairs minimax path problem (Hu [1961])
4. Reducing data storage (Kang, Lee, Chang, and Chang [1977])
5. Cluster analysis (Gower and Ross [1969], and Zahn [1971])

In Application 1.7 we described another application of the spanning tree problem that arises in measuring the homogeneity of bimetallic objects (Shier [1982] and Filliben, Kafadar, and Shier [1983]). Additional applications of the minimum spanning tree problem arise in (1) solving a special case of the traveling salesman problem (Gilmore and Gomory [1964]), (2) chemical physics (Stillinger [1967]), (3) Lagrangian relaxation techniques (Held and Karp [1970]), (4) network reliability analysis (Van Slyke and Frank [1972]), (5) pattern classification (Dude and Hart [1973]), (6) picture processing (Osteen and Lin [1974]), and (7) network design (Magnanti and Wong [1984]). The survey paper of Graham and Hell [1985] provides references for additional applications of the minimum spanning tree problem.

EXERCISES

13.1. Suppose that you want to determine a spanning tree T that minimizes the objective function $[\sum_{(i,j)\in T} (c_{ij})^2]^{1/2}$. How would you solve this problem?

13.2. In the network shown in Figure 13.14, the bold lines represent a minimum spanning tree.

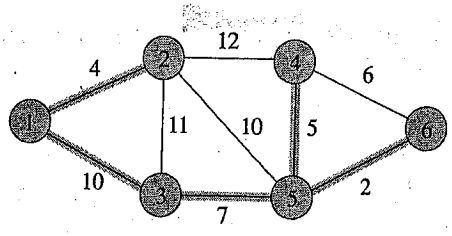


Figure 13.14 Verifying cut and path optimality conditions.

(a) By listing each nontree arc (k, l) and the minimum length arc on the tree path from node k to node l, verify that this tree satisfies the path optimality conditions.

(b) By listing each tree arc (i, j) and the minimum length arc in the cut defined by the arc (i, j), verify that the tree satisfies the cut optimality conditions.

13.3. Using Kruskal's algorithm, find minimum spanning trees of the graphs shown in Figure 13.15.

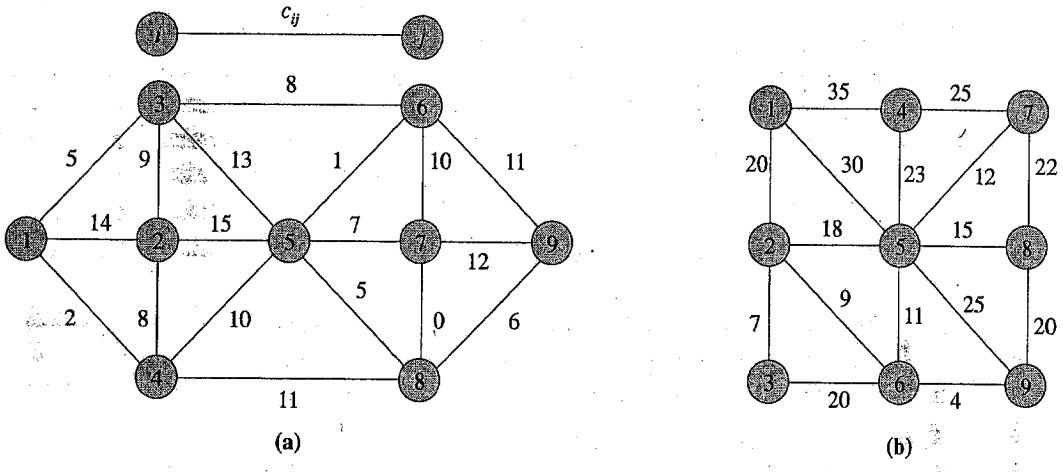


Figure 13.15 Examples for Exercises 13.3 to 13.5.

13.4. Using Prim's algorithm, find minimum spanning trees of the graphs shown in Figure 13.15.

13.5. Using Sollin's algorithm, find minimum spanning trees of the graphs shown in Figure 13.15.

13.6. Think of the network shown in Figure 13.16 as a highway map, and the number recorded next to each arc as the maximum elevation encountered in traversing the arc. A traveler plans to drive from node 1 to node 12 on this highway. This traveler dislikes high altitudes and so would like to find a path connecting node 1 to node 12

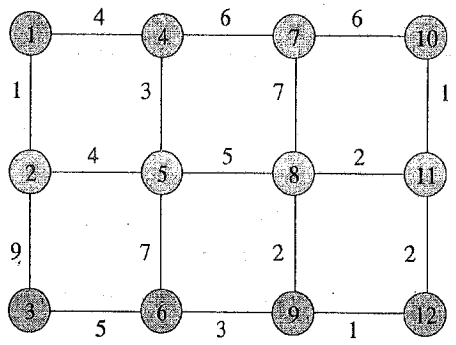


Figure 13.16 Highway grid.

that minimizes the maximum altitude. Find the best path for this traveler using a minimum spanning tree algorithm.

13.7. Can you generalize the approach outlined in Application 13.3 to solve the all-pairs maximum capacity path problem in directed networks? If yes, describe your algorithm; if not, why not?

13.8. In Theorem 13.3 we proved the sufficiency of the path optimality conditions using the cut optimality conditions. Give a direct proof of this sufficiency condition that does not use the cut optimality conditions.

13.9. Prove the maximum spanning tree optimality conditions stated in Theorem 13.4.

13.10. Let (p, q) be a minimum cost arc in G. Show that (p, q) belongs to some minimum spanning tree of G. Does every minimum spanning tree of G contain the arc (p, q)?

13.11. Show that a maximum weight acyclic subgraph in an undirected graph G with strictly positive arc weights c_{ij} must be a spanning tree.

13.12. How would you modify Kruskal's and Prim's algorithms to solve the maximum spanning tree problem?

13.13. In an undirected network, we define a tree of shortest paths as a spanning tree in which the unique path from a specified node s to every other node is a shortest path. Is a minimum spanning tree of G also a tree of shortest paths? Either prove this result or construct an example to show that the trees could be different.

13.14. Tree minimax result. Let $G = (N, A)$ be an undirected network with a capacity u_{ij} associated with every arc $(i, j) \in A$. For any spanning tree T of G, we define its *capacity* as $\min\{u_{ij}:(i, j) \in T\}$, and for any cut Q of G, we define its *value* as $\max\{u_{ij}:(i, j) \in Q\}$. Show that the capacity of any spanning tree is a lower bound on the value of every cut. Next show that the maximum capacity of any spanning tree equals the minimum value of any cut.

13.15. We say that two spanning trees T' and T'' are *adjacent* if they have all but one arc in common. Show that for any two spanning trees T' and T'', we can find a sequence of spanning trees $T^1, T^2, \ldots, T^k$ with $T^1 = T'$, $T^k = T''$ and with T^i adjacent to T^{i+1} for every $i = 1$ to $k - 1$.

13.16. Suppose that you are given a graph with each arc colored either red or blue.

(a) Show how to find a spanning tree with the maximum number of red arcs.

(b) Suppose that some spanning tree has k' red arcs and another spanning tree has $k'' > k'$ red arcs. Show that for every k, $k' \leq k \leq k''$, some spanning tree has k red arcs.

13.17. Let T be a spanning tree. For any pair $[i, j]$ of nodes, let $\beta[i, j]$ denote the least cost arc among the arcs in the tree path joining node i and node j. Show how to compute $\beta[i, j]$ for every pair of nodes in a total of $O(n^2)$ time.

13.18. In a class of undirected networks, suppose that all arc costs are *small* (i.e., they lie in the interval $[1, k]$ for some small integer k, say $k = 10$). How fast could you implement Kruskal's and Prim's algorithms for solving the minimum spanning tree problem in this class of networks?

13.19. Consider the following *reverse greedy algorithm*:

```
begin
  let the arcs (i1, j1), (i2, j2), ..., (im, jm) be arranged in
    nonincreasing order of their lengths;
  G' := G;
  for k := 1 to m do
    if G' - {(ik, jk)} is a connected graph then
      G' := G' - {(ik, jk)};
end;
```

Show that at the termination of this algorithm, the graph G' is a minimum spanning tree.

13.20. Consider the following algorithm. Arrange the arcs in A in any arbitrary order and start with a null tree T. Examine each arc (i, j) in A, one by one, and perform the following steps: add arc (i, j) to T and if T contains a cycle W, delete from T an arc of maximum cost from the cycle W. Show that when this algorithm has examined all the arcs, the final tree T is a minimum spanning tree. Is it possible to implement this algorithm as efficiently as Kruskal's algorithm? Why or why not?

13.21. Can you use the data structure of Dial's implementation of Dijkstra's shortest path algorithm (discussed in Section 4.6) to implement Prim's algorithm? If so, is the running time of Prim's algorithm better than the running time of the shortest path algorithm?

13.22. In Section 13.5 we observed a striking resemblance between Prim's algorithm and Dijkstra's algorithm. This observation might lead us to conjecture that we can use a radix heap data structure (discussed in Section 4.8) to implement Prim's algorithm in $O(m + n \log(nC))$ time. However, this conjecture is not valid. What are the difficulties we would encounter if we attempted to implement Prim's algorithm using radix heaps?

13.23. The first implementation of Kruskal's algorithm that we discussed in Section 13.4 selects a nontree arc (k, l) violating its optimality condition and exchanges this arc with some tree arc of lower cost. Show that no matter which order we use to select the nontree arcs violating their optimality conditions, we perform at most nm iterations. (*Hint*: Let $f(i, j)$ be the number of arcs in the network whose cost is strictly greater than c_{ij}. Consider the effect on the potential function $= \sum_{(i,j)\in T} f(i, j)$ as we change the spanning tree T.)

13.24. Let T be a minimum spanning tree of an undirected graph $G = (N, A)$ and let Q be a set of nontree arcs (k, l) satisfying the following property: Some arc (i, j) in the tree path from node k to node l has the same cost as arc (k, l); that is, $c_{ij} = c_{kl}$. Professor May B. Wright claims that every spanning tree in the subgraph $G' = (N, T \cup Q)$ is a minimum spanning tree of G. Construct a counterexample to show that Professor Wright's claim is false.

13.25. Sensitivity analysis. Let T^* be a minimum spanning tree of a graph $G = (N, A)$. For any arc $(i, j) \in A$, we define its *cost interval* as the set of values of c_{ij} for which T^* continues to be a minimum spanning tree.

(a) Describe an efficient method for determining the cost interval of a given arc (i, j). (*Hint*: Consider two cases: When $(i, j) \in T^*$ and when $(i, j) \notin T^*$, and use the cut and path optimality conditions.)

(b) Describe a method for determining the cost intervals of every arc in A. Your method must be faster than determining the cost intervals of each arc one by one. (*Hint*: Use the result of Exercise 13.17.)

13.26. Suppose that we have in hand a minimum spanning tree T for the undirected graph $G = (N, A)$. Suppose that we add a new node $(n + 1)$ to N and p new arcs to A incident to this node. How fast can you find a minimum spanning tree for the enlarged network G' from the minimum spanning tree T of G? (*Hint*: Use the cut optimality conditions.)

13.27. Arc additions and deletions. Let T^* be a minimum spanning tree for the undirected graph $G = (N, A)$. Describe an algorithm for reoptimizing the minimum spanning tree when we delete an arc $(i, j) \in A$ from the network. Similarly, describe an algorithm for reoptimizing the problem when we add a new arc (i, j) to A. Prove that your algorithms correctly find new minimum spanning trees and state their running times.

13.28. Spanning trees containing specific arcs. In an undirected graph $G = (N, A)$, let (p, q) be a specified arc. Describe a method for identifying a minimum spanning tree T^* subject to the condition that the tree must contain the arc (p, q). Prove that your method correctly solves this problem. Generalize the method for situations in which the minimum spanning tree must contain an acyclic set A' of arcs. (*Hint*: Assign appropriate costs to the arcs required to be in the optimal tree.)

13.29. Factored minimum spanning tree problem. Let G be a complete undirected graph. Suppose that we associate a positive real number α_i with each node $i \in N$ and define

the cost of each arc (i, j) as $c_{ij} = \alpha_i \alpha_j$. This specialized minimum spanning tree problem is known as the *factored minimum spanning tree problem*. We wish to develop an algorithm for solving this class of problems that is more efficient than the general minimum spanning tree algorithms.

(a) Consider a five-node network with $\alpha_i = i$. Find a minimum spanning tree in this network.

(b) Use the insight obtained from answering part(a) to develop an $O(n)$ algorithm for solving the factored minimum spanning tree problem.

13.30. Most vital arcs. In the minimum spanning tree problem, we refer to an arc as a *vital arc* if its deletion strictly increases the cost of the minimum spanning tree. A *most vital arc* is a vital arc whose deletion increases the cost of the minimum spanning tree by the maximum amount.

(a) Does a network always contain a vital arc?

(b) Suppose that a network contains a vital arc. Describe an $O(nm)$ algorithm for identifying a most vital arc. Can you develop an algorithm that runs faster than $O(nm)$ time? (*Hint*: Use the cut optimality conditions.)

13.31. Suppose that we arrange all the spanning trees of a graph G in nondecreasing order of their costs. We refer to a spanning tree T as *a* kth *minimum spanning tree* if it is at the kth position in this order. Describe an $O(n^2)$ algorithm for finding the second minimum spanning tree. (*Hint*: Observe that the second minimum spanning tree must contain at least one arc that is not in the first minimum spanning tree. Then use the result of Exercise 13.17.)

13.32. Bottleneck spanning trees. A spanning tree T is a *bottleneck spanning tree* if the maximum arc cost in T is as small as possible from among all spanning trees. Show that a minimum spanning tree of G is also a bottleneck spanning tree of G. Is the converse result also true (i.e., is a bottleneck spanning tree of G also a minimum spanning tree of G)? Either prove this result or construct a counterexample.

13.33. Describe an $O(m \log n)$ algorithm, using binary search, for solving the bottleneck spanning tree problem defined in Exercise 13.32.

13.34. Balanced spanning trees. A spanning tree T is a *balanced spanning tree* if from among all spanning trees, the difference between the maximum arc cost in T and the minimum arc cost in T is as small as possible. Describe an $O(m^2)$ algorithm for determining a balanced spanning tree.

13.35. Parametric analysis of minimum spanning trees. In the parametric minimum spanning tree problem, each arc length $c_{ij} = c_{ij}^0 + \lambda c_{ij}^*$ is a linear function of a parameter λ. Let T^λ denote a minimum spanning tree with arc lengths chosen as $c_{ij}^0 + \lambda c_{ij}^*$ for a specific value of λ.

(a) Show that for sufficiently large values of the constant $k > 0$, T^{-k} and T^k are the maximum and minimum spanning trees when the arc lengths are c_{ij}^*.

(b) Show that T^λ is a minimum spanning tree for all of the values of λ in some interval $[\underline{\lambda}, \bar{\lambda}]$. Moreover, show that at the lower and upper limits of this interval, at least two alternate minimum spanning trees are adjacent in the sense of Exercise 13.15. (*Hint*: Use the path optimality conditions.)

(c) Describe an algorithm for determining a minimum spanning tree for all values of λ from $-\infty$ to $+\infty$.

13.36. (a) Show that in the parametric minimum spanning tree problem, as we vary λ from $-\infty$ to $+\infty$, we obtain at most m^2 minimum spanning trees and every two consecutive minimum spanning trees are adjacent. (*Hint*: Use the fact that if T' and T'' are two consecutive minimum spanning trees, we can obtain T'' from T' by replacing a tree arc (i, j) by a nontree arc (k, l) satisfying the condition $c_{kl}^* \leq c_{ij}^*$.)

(b) Consider a special case of the parametric minimum spanning tree problem in which each $c_{ij}^* = 0$ or 1. Show that in this case, as we vary λ from $-\infty$ to $+\infty$, we obtain at most n minimum spanning trees.

(c) Consider another special case of the parametric minimum spanning tree problem in which each $c^*_{pq} = 1$ for a specific arc (p, q) and is zero for all other arcs. Show how to find minimum spanning trees for all values of λ in time proportional to solving a single minimum spanning tree problem.

(d) Consider yet another special case of the minimum spanning tree problem where all parametric arcs are incident to a common node p (i.e., $c^*_{ij} = 1$ whenever $i = p$ or $j = q$, and is zero for all other arcs). How fast can you find minimum cost spanning trees for all values of λ?

13.37. Minimum ratio spanning trees (Chandrasekaran [1977]). In the minimum ratio spanning tree problem, we associate two numbers, c_{ij} and τ_{ij}, with each arc (i, j) in a network G and wish to determine a spanning tree T^* that minimizes $(\sum_{(i,j)\in T} c_{ij})/(\sum_{(i,j)\in T^*} \tau_{ij})$ from among all spanning trees. We assume that $\sum_{(i,j)\in T^*} \tau_{ij} > 0$ for all spanning trees T. Suggest a binary search algorithm for identifying a minimum ratio spanning tree of G that runs in polynomial time.

13.38. A *1-tree* of G is a spanning tree of G plus one arc. Show that the minimum spanning tree of G plus the least cost nontree arc defines a minimum cost 1-tree of G. Suppose that the additional arc must be adjacent to a particular node s of G. How would you find a minimum cost 1-tree for this version of the problem?

13.39. Optimal 1-forest. A set of arcs is a *1-forest* of an undirected graph G if some arc (k, l) in F satisfies the condition that $F - \{(k, l)\}$ is a forest.

(a) Show that the collection of all 1-forests forms a matroid.

(b) Give a greedy algorithm for identifying a maximum weight 1-forest of G.

(c) How would you modify your answers to parts (a) and (b) if we required that the arc (k, l) be incident to a specific node s of the network?

13.40. Optimal k-forest. A set F of arcs is a k-forest of an undirected graph G if some subset $F' \subseteq F$ containing k arcs satisfies the condition that $F - F'$ is a forest. Show that the collection of all k-forests forms a matroid and give a greedy algorithm for identifying a maximum weight k-forest of G. (*Hint*: Generalize the result in Exercise 13.39.)

13.41. Let F_p and F_{p+1} be forests in a graph containing p and $p + 1$ arcs. Show that we can always add some arc in F_{p+1} to F_p to produce a forest with $p + 1$ arcs.

13.42. Using the example we have considered in the text as motivation, give a formal proof of Theorem 13.9.

13.43. Linear programming proof of the greedy algorithm. Let $(E, \mathcal{I})$ be a matroid with an associated weight w_e for $e \in E$. Let x_e be a zero–one vector indicating whether or not the element e is a member of a set I from E; that is, $x_e = 1$ if $e \in I$ and $x_e = 0$ if $e \notin I$. For any subset S in E, let $r(S)$ denote its rank, defined as the number of elements of the largest independent set in S. For example, the rank of a set S of arcs in a graph is the size of the largest forest defined by these arcs.

(a) Show that the incidence vectors x_e of a basis of the matroid satisfy the following conditions:

$$\sum_{e\in E} x_e = r(E), \tag{13.3a}$$

$$\sum_{e\in S} x_e \le r(S) \qquad \text{for all } S \subseteq \mathcal{I}, \tag{13.3b}$$

$$x_e \ge 0. \tag{13.3c}$$

(b) Show that for the minimum spanning tree problem, the constraints in (13.3) contain all of the constraints in the formulation (13.2).

(c) Mimicking the proof of Theorem 13.9 (see Exercise 13.42), give a linear programming proof that the greedy algorithm solves the matroid optimization problem of finding a basis of the matroid $(E, \mathcal{I})$ with the smallest possible weight $w(B)$.

13.44. Linear programming formulation of matroids. In Theorem 13.10 we showed that spanning trees of a graph correspond to the extreme points of the linear program (13.2).

Using the result of Exercise 13.43, show that the bases of a matroid correspond to the extreme points of the polyhedron defined by the constraints given in (13.3). (*Hint*: Use the result of Exercise 13.43 and the fact that each extreme point of a linear program is the unique optimal solution for some choice of the objective coefficients.)

13.45. In Exercise 13.43 we showed that the greedy algorithm solves the matroid optimization problem. Show that this property actually characterizes matroids. That is, show that the greedy algorithm will solve the minimum weight independent set problem for any choice of the element weights if and only if the subset system is a matroid. (*Hint*: Any subset system that is not a matroid contains two independent subsets I and I' satisfying the property that $|I'| > |I|$ and no element in I' can be added to I to obtain an independent set. Define the weight function on E appropriately so that the greedy algorithm terminates with I, but I' is optimal.)

13.46. **(a)** The set of minimum spanning trees $T^1, T^2, \ldots, T^{k-1}$ that we determined in Exercise 13.36(d) as we varied the parameter from $C + 1$ to $-\infty$ satisfy the "monotonicity" property that once an arc $(1, j)$ belongs to any tree T^p, it also belongs to all of the trees T^q for $q \geq p$. Suppose that the parametric cost of arc $(1, j)$ is $c_{1j} + \lambda d_{1j}$ for some constant d_{1j} and that the cost of arc (i, j) is c_{ij} for $i \neq 1$ and $j \neq 1$. Does the set of optimal spanning trees, as we vary λ from $C + 1$ to $-\infty$, satisfy the monotonicity property?

(b) If possible, describe a polynomial time variant of the procedure discussed in Exercise 13.36(d) that will solve the parametric problem defined in part (a). If you cannot describe any such algorithm, explain the difficulties encountered.

14

CONVEX COST FLOWS

I bend but do not break.
—Jean de La Fontaine

Chapter Outline

14.1 INTRODUCTION

Essentially all fields of scientific inquiry evolve into specialized branches of investigations, each with its own particular traditions and approaches. The field of optimization is no exception; it divides quite naturally into several ways: constrained versus unconstrained optimization; linear versus nonlinear programming; and discrete versus continuous optimization. By its very nature, network optimization is a special class of constrained optimization problems. We can, however, distinguish the various domains of network optimization along the other dimensions. Our development has focused exclusively on linear models. Moreover, because the integrality property ensures that linear minimum cost flow problems, even when stated as continuous optimization models, always have integer solutions (assuming that the data are integral), with the exception of the matching and spanning tree problems that we have considered in the preceding two chapters, we have not had to make any distinction between discrete and continuous models. Yet many of the arguments and approaches in network optimization have a distinct combinatorial flavor. Indeed, the optimization community typically views network flows as the starting point for building much of the theory and algorithmic approaches of discrete optimization.

Suppose that we wish to extend our discussion into the realm of nonlinear optimization. What type of models should we consider? Perhaps the most natural approach would be to replace the linear objective function of the minimum cost flow problem by a general nonlinear function. Although we might like to consider the most general nonlinear functions possible, doing so would take us far afield from the mainstream of our investigations. Instead, we might ask the following question: Is there a class of nonlinear optimization models that arise frequently in practice and that we can solve by adapting the algorithmic approaches that we have already

developed? In this chapter we examine one such set of models, those with separable convex objective functions. Fortunately, these models provide us with the most useful set of nonlinear objective functions that arise in the practice of network optimization. Moreover, this particular set of models permits us to remain within the domain of discrete optimization, since as we will see in this chapter, if we further restrict these models by requiring the solutions to be integer, we can solve these problems quite efficiently, in theory as well as in practice.

In all the models we have considered to this point, the objective function was separable in the sense that the different flow variables x_{ij} appeared in separate terms $c_{ij}x_{ij}$. In the models we consider in this chapter, we retain the separability assumption, but we now permit the separable terms to be nonlinear functions of the form $C_{ij}(x_{ij})$. We also impose an additional convexity assumption: Each function $C_{ij}(x_{ij})$ is convex: the functions are "bathtub" shaped in the sense that linear interpolations always lie on or above the functions (mathematically, if θ is a parameter satisfying $0 \le \theta \le 1$ and x'_{ij} and x''_{ij} are any two points within the flow bounds of x_{ij}, then $C_{ij}(\theta x'_{ij} + (1 - \theta)x''_{ij}) \le \theta\, C_{ij}(x'_{ij}) + (1 - \theta)C_{ij}(x''_{ij}))$. Figure 14.1 gives two examples of convex functions.

We consider two different models:

1. *Piecewise linear model* [see Figure 14.1(a)]. Each arc cost $C_{ij}(x_{ij})$ has at most p linear segments: $0 = d_{ij}^0 < d_{ij}^1 < d_{ij}^2 < \cdots$ denote the breakpoints of the function and the cost varies linearly in the interval d_{ij}^{k-1} to d_{ij}^k. We let c_{ij}^k denote the linear cost coefficient in the interval $[d_{ij}^{k-1}, d_{ij}^k]$. Therefore, to specify a piecewise linear cost function, we need to specify the breakpoints and the slopes of the linear segments between successive breakpoints.
2. *Concise function model* [see Figure 14.1(b)]. The functions $C_{ij}(x_{ij})$ are specified in a functional form, such as x_{ij}^4. In this case we often require only $O(1)$ information to specify the function. For this model, we assume that we restrict the feasible solutions to integers. Although we could easily adapt the algorithm

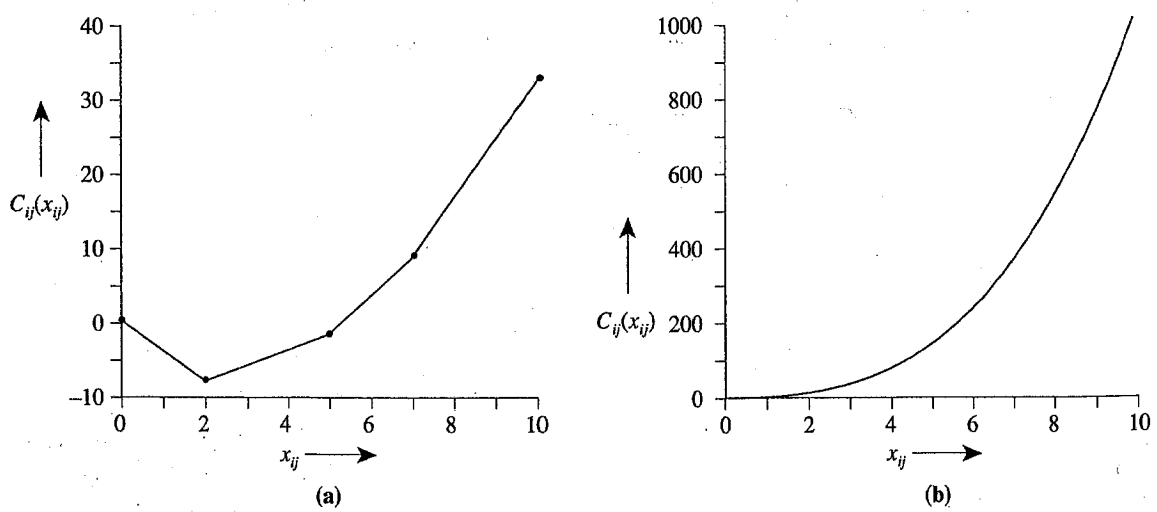


Figure 14.1 Two examples of convex cost flow functions.

that we examine for this model to solve continuous problems, our computational complexity results apply only to the integer model.

The integer restriction on the optimal solution does impose some loss of generality because the integer optimal solution might not be as good as a continuous optimal solution. But we can always obtain an integer optimal solution as close as desired to a continuous optimal solution by scaling the data. For example, if we want a solution more accurate than the integer optimal solution, we could substitute My_{ij}, for sufficiently large value of M, for each x_{ij}. We would chose M depending on the accuracy we desired (e.g., $M = 1{,}000$ or $10{,}000$). If y_{ij}^* denotes an integer optimal solution of the transformed problem, $x_{ij}^* = y_{ij}^*/M$ is an optimal solution of the original problem (to a degree of accuracy of $1/M$). This technique allows us to obtain a real-valued optimal solution of the convex cost flow problem to any desired degree of accuracy.

Note that in view of the integrality assumption, we can assume that each convex cost function is a piecewise linear function since we can allow each integer point to be a breakpoint of the function and linearize the function between these breakpoints (see Figure 14.2). However, we differentiate between the two models we have introduced for the following reason:

1. When we specify the function by specifying the breakpoints and the slopes of the function between successive breakpoints, the length of the input data is proportional to the number of linear segments in all the cost functions.
2. When we specify the function concisely, we assume the length of the input data for arc (i, j) is $O(1)$, but when we linearize it, it will have U segments, where U is the largest arc capacity. In this case the length of the input data is not proportional to the total number of segments.

It is necessary to distinguish between these two cases because an algorithm that solves breakpoint problems in polynomial time might not solve the concise-function model in polynomial time.

Before describing algorithms for solving these problems, we discuss several applications of the convex cost model. First, however, let us formally define the

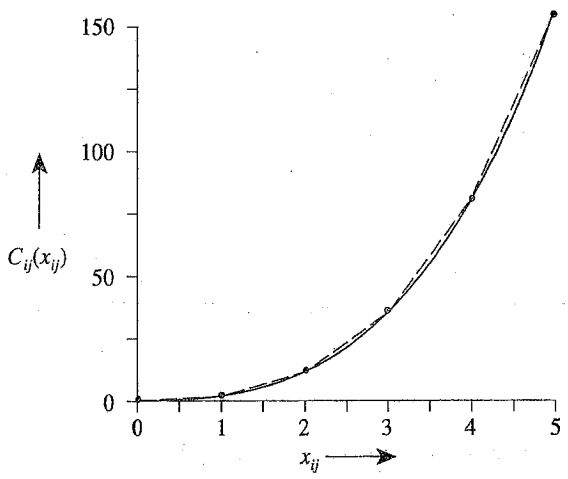


Figure 14.2 Transforming a function in concise form to a piecewise linear form. The dashed line shows the piecewise linear approximation.

convex cost model that we will be considering and introduce several assumptions that we will be imposing. We formulate the model as the following optimization problem:

$$\text{Minimize} \quad \sum_{(i,j)\in A} C_{ij}(x_{ij}) \tag{14.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i) \qquad \text{for all } i \in N, \tag{14.1b}$$

$$0 \le x_{ij} \le u_{ij} \qquad \text{for all } (i, j) \in A, \tag{14.1c}$$

$$x_{ij} \text{ is integer for all } (i, j) \in A. \tag{14.1d}$$

We define this model on a directed network $G = (N, A)$ with a capacity u_{ij} and a convex cost function $C_{ij}(x_{ij})$ associated with every arc $(i, j) \in A$. As always, we associate a number $b(i)$ with each node $i \in N$ specifying the node's supply or demand, depending on whether $b(i) > 0$ or $b(i) < 0$. Let U denote the largest number among the supplies/demands of the nodes and the finite arc capacities.

We impose several assumptions that we discussed in some detail in Section 9.1 for the minimum cost flow problem: (1) the network is directed; (2) all the supply/demand values $b(i)$ are integers; $\sum_{i\in N} b(i) = 0$; and the convex cost flow problem has a feasible solution; (3) the lower bounds on all the arc flows are zero; and (4) the network contains a directed uncapacitated path between every pair of nodes. Using arguments similar to those that we have used for the minimum cost flow problem, we can show that we incur no loss of generality by imposing these assumptions. We also assume that for each arc (i, j), $C_{ij}(x_{ij}) = 0$ when $x_{ij} = 0$. This assumption imposes no loss of generality because we can always satisfy it by defining $C'_{ij}(x_{ij}) = C_{ij}(x_{ij}) - C_{ij}(0)$.

14.2 APPLICATIONS

Many of the linear network flow models that we have examined in our previous discussions have rather natural nonlinear cost generalizations. System congestion and queuing effects are one source of these nonlinearities (since queuing delays vary nonlinearly with flows). In finance, we often are interested in not only the returns on various investments, but also in their risks, which analysts often measure by quadratic functions. In some other applications, cost functions assume different forms over different operating ranges, so the resulting cost function is piecewise linear. For example, in production applications, the cost of satisfying customers' demand is different if we meet the demand from current inventory or by backordering items.

To give a flavor of the applications of convex cost network flow models, in this section we describe four applications. The first one is a direct application of physical systems' nonlinear cost imposed upon the flows. The second application is one in which different operating ranges produce different costs. In the last two applications, even though the underlying problem is not a flow problem, we can model it as a convex cost network flow model.

Application 14.1 Urban Traffic Flows

In road networks, as more vehicles use any road segment, the road becomes increasingly congested and so the delay on that road increases. For example, the delay on a particular road segment, as a function of the flow x on that road, might be $\alpha x/(u - x)$. In this expression u denotes a theoretical capacity of the road and α is another constant: As the flow increases, so does the delay; moreover, as the flow x approaches the theoretical capacity of that road segment, the delay on the link becomes arbitrarily large. In many instances, as in this example, the delay function on each road segment is a convex function of the road segment's flow, so finding the flow plan that achieves the minimum overall delay, summed over all road segments, is a convex cost network flow model.

Another model of urban traffic flow rests on the behavioral assumption that users of the system will travel, with respect to any prevailing system flow, from their origin to their destination by using a path with the minimum delay. So if $C_{ij}(x_{ij})$ denotes the delay on arc (i, j) as a function of the arc's flow x_{ij}, each user of the system will travel along a shortest delay path with respect to the total delay cost $C_{ij}(x_{ij})$ on the arcs of that path. Note that this problem is a complex equilibrium model because the delay that one user incurs depends on the flow of other users, and all of the users are simultaneously choosing their shortest paths. In this problem setting, we can find the equilibrium flow by solving a convex network flow model with the objective function

$$\sum_{(i,j)\in A} \int_0^{x_{ij}} C_{ij}(y)\, dy.$$

If the delay function is nondecreasing, the function of each variable x_{ij} within the summation is convex, and since the sum of convex functions is convex (see Exercise 14.1), the overall objective function is convex. Moreover, if we solve the network optimization problem defined by this objective function and the network flow constraints, the optimality conditions are exactly the shortest path conditions for the users. (See the reference notes for the details of these claims.)

This example is a special case of a more general result, known as a *variational principle*, that arises in many settings in the physical and social sciences. The variational principle says that to find an equilibrium of a system, we can solve an associated optimization problem: The optimality conditions for the problem are then equivalent with the equilibrium conditions.

Application 14.2 Area Transfers in Communication Networks

In communication networks, telephones do not have sufficient "intelligence" to route calls between each other (historically, equipping every telephone with the ability to route its own calls has been prohibitively expensive). Instead, the system connects each of the telephones within a collection of customers directly to a sophisticated telecommunication device known as a *switching center*; this center does all of the routing for the telephones that "home into" it. That is, the switching center receives and sends all of the calls (1) between its assigned customers, and (2) between

these customers and every other customer in the system (who home into some other switching center). Because the switching centers have limited capacity, as communication traffic in the system increases, a telephone company must either add capacity at one or more of its centers or make "area transfers," that is, rehome traffic from one switching center to another.

Consider a communication network with regions divided into many districts that are served by several switching centers. Let $d(j)$ denote the current demand of district j, as measured by number of lines, and let $b(i)$ denote the capacity of switching center i, that is, the maximum number of lines the switching center can handle. To meet the current demands of the districts, the company currently uses w_{ij} working lines between switching center i and district j. To satisfy future demands for lines at district j, the company can use some of the s_{ij} spare lines connecting switching center i to district j at a cost of λ_{ij} per line. It can also add additional lines beyond the available spares at a larger per line cost of $\delta_{ij} > \lambda_{ij}$. To avoid exceeding the capacity at switching center i the company can make "area transfers" by disconnecting a line connecting switching center i to district j, at a cost of μ_{ij} per line, and reconnecting the district to another switching center. The company faces the following problem. Given the new demands $d(j) + \Delta(j)$ for lines at each district j, how should it assign the customers in the districts to the switching centers (with possible area transfers) at the least possible total cost?

Figure 14.3(a) shows a network formulation for this problem. The flow x_{ij} on the arc from the switching center i to district j represents the capacity of the switching center i allocated to district j. Figure 14.3(b) specifies the cost of the flow on arc (i, j). Notice that if switching center i supplies w_{ij} lines to district j, it incurs no additional cost; supplying any less of an allocation than w_{ij} incurs the costs of area transfers, and supplying any more allocation than w_{ij} incurs the cost of adding lines. Because of the area transfer cost and the added incremental costs of using spare lines, the cost structure is nonlinear, so the problem is a convex cost network flow model.

Application 14.3 Matrix Balancing

Statisticians, social scientists, agricultural specialists, and many other practitioners frequently use experimental data to try to draw inferences about the effects of certain control parameters at their disposal (e.g., the effects of using different types of fertilizers). These practitioners often use contingency tables to classify items according to several criteria, with each criterion partitioned into a finite number of categories. As an example, consider classifying r individuals in a population according to the criteria of marital status and age, assuming that we have divided these two criteria into p and q categories, respectively. As categories for marital status, we might choose single, married, separated, or divorced; as categories for age, we might choose below 20, 21–25, 26–30, and so on. This categorization gives a table of pq cells. Dividing the entry in each cell by the number of items r (i.e., the total population) gives the (empirical) probability of that cell.

In many applications it is important to estimate the current cell probabilities, which change continually with time. We can estimate these cell probabilities very accurately using a census, or approximately by using a statistical sampling of the

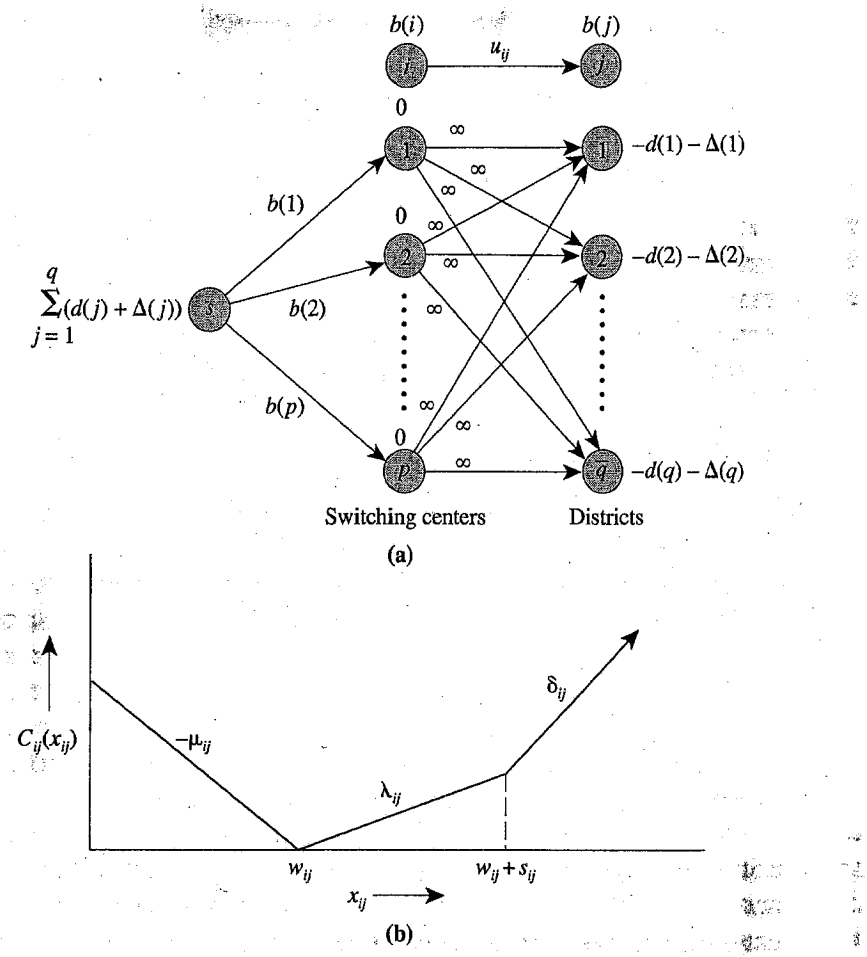


Figure 14.3 (a) Network for the area transfers in communication networks; (b) cost structure of arcs from switching centers to districts.

population; however, even this sampling procedure is expensive. Typically, we would calculate the cell probabilities by sampling only occasionally (for some applications, only once in several years), and at other times revise the most recent cell probabilities based on partial observations. Suppose that we let α_{ij} denote the most recent cell probabilities. Suppose, further, that we know some aggregate data in each category with high precision; in particular, suppose that we know the row sums and column sums. Let u_i denote the number of individuals in the ith marital category and let v_j denote the number of individuals in the jth age category. Let $r = \sum_{i=1}^{p} u_i$. We want to obtain an estimate of the current cell probabilities x_{ij}'s so that the cumulative sum of the cell probabilities for the ith row equals u_i/r, the cumulative sum of the cell probabilities for the jth column equals v_j/r, and the matrix x is, in a certain sense, *nearest* to the most recent cell probability matrix α. One popular measure of defining the *nearness* is to minimize the weighted cumulative squared deviation of the individual cell probabilities. With this objective, our problem reduces to the following convex cost flow problem:

$$\text{Minimize} \quad \sum_{i=1}^{p} \sum_{j=1}^{q} w_{ij}(x_{ij} - \alpha_{ij})^2 \tag{14.2a}$$

subject to

$$\sum_{j=1}^{q} x_{ij} = u_i/r \qquad \text{for all } i = 1, \ldots, p, \tag{14.2b}$$

$$-\sum_{i=1}^{p} x_{ij} = -v_j/r \qquad \text{for all } j = 1, \ldots, q, \tag{14.2c}$$

$$x_{ij} \geq 0 \qquad \text{for all } i = 1, \ldots, p, \text{ and for all } j = 1, \ldots, q. \tag{14.2d}$$

This type of *matrix balancing problem* arises in many other application settings. The interregional migration of people provides another important application. In the United States, using the general census of the population, taken once every 10 years, the federal government produces flow matrices with detailed migration characteristics. It uses these data for a wide variety of purposes, including the allocation of federal funds to the states. Between the 10-year census, net migration estimates for every region become available as by-products of annual population estimates. Using this information, the federal government updates the migration matrix so that it can reconcile the out-of-date detailed migration patterns with the more recent net figures.

Application 14.4 Stick Percolation Problem

One method for improving the structural properties of (electrically) insulating materials is to embed sticks of high strength in the material. The current approach used in practice is to add, at random, sticks of uniform length to the insulating material. Because the sticks are generally conductive, if they form a connected path from one side of the material to the other, the configuration will destroy the material's desired insulating properties. Material scientists call this phenomenon *percolation*. Although longer sticks offer better structural properties, they are more likely to cause percolation. Using simulation, analysts would like to know the effect of stick length on the possibility that a set of sticks causes percolation, and when percolation does occur, the resulting heat loss due to (electrical) conduction.

Analysts use simulation in the following manner: The computer randomly places p sticks of a given length L in a square region; see Figure 14.4(a) for an example. We experience heat loss because of the flow of current through the intersection of two sticks. We assume that each such intersection has unit resistance. We can identify whether percolation occurs and determine the associated power dissipation by creating an equivalent resistive network as follows. We assign a resistor of 1 unit to every intersection of the sticks. We also associate a current source of 1 unit with one of the boundaries of the insulant and a unit sink with the opposite boundary. The problem then is to determine the power dissipation of the resistive network.

Figure 14.4(b) depicts the transformation of the stick percolation problem into a network model. In this network model, each node represents a resistance and contributes to the power dissipation. The node splitting transformation described in Section 2.4 permits us to model the node resistances as arc resistances. Recall from Ohm's law that a current flow of x amperes across a resistor of r ohms creates a power dissipation of rx^2 watts. Moreover, the current flows in an electrical network

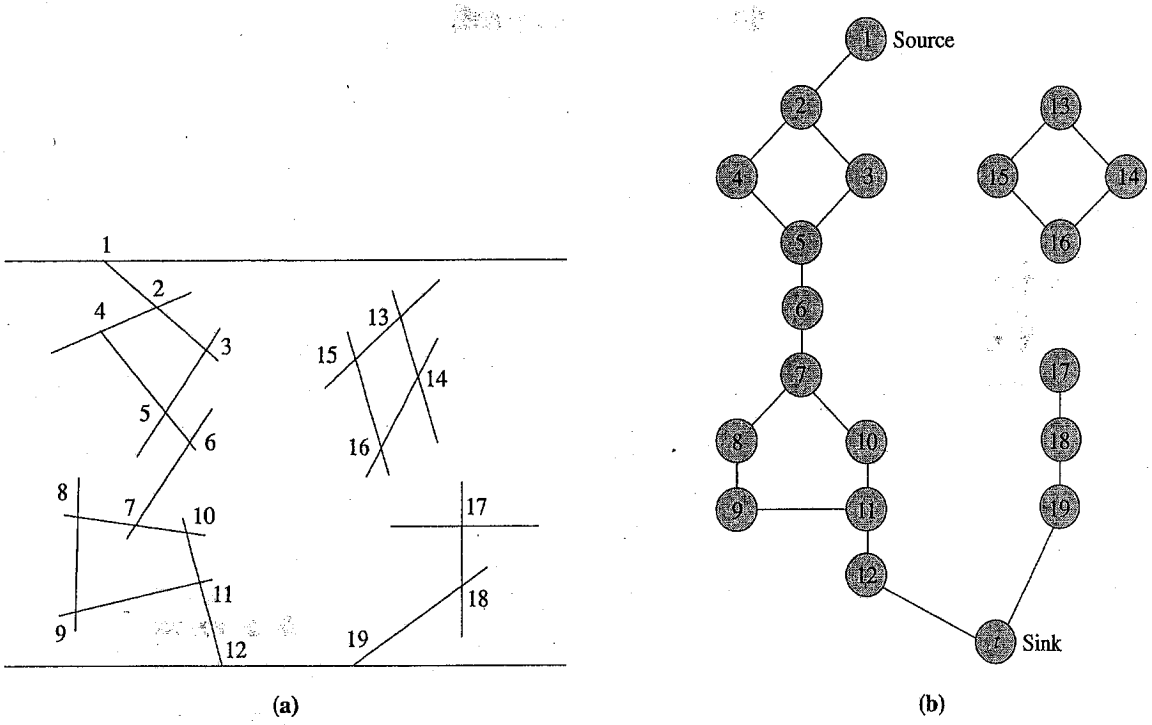


Figure 14.4 Formulating the stick percolator problem: (a) placement of sticks; (b) corresponding network.

in a way that minimizes the rate of power dissipation (i.e., follows the path of least resistance). Consequently, we can state the stick percolator problem as the following convex cost flow problem.

$$\text{Minimize} \quad \sum_{(i,j)\in A} r_{ij} x_{ij}^{k} \tag{14.3a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ji} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} 1 & \text{for } i = s \\ 0 & \text{for all } i \in N - \{s, t\}, \\ -1 & \text{for } i = t \end{cases} \tag{14.3b}$$

$$x_{ij} \geq 0 \qquad \text{for all } (i, j) \in A. \tag{14.3c}$$

In this model, x_{ij} is the current flow on arc (i, j). The solution of this convex cost flow model indicates whether percolation occurs (i.e., if the problem has a feasible solution), and if so, the solution specifies the value of the associated power loss.

14.3 TRANSFORMATION TO A MINIMUM COST FLOW PROBLEM

In this section we show how to transform a convex cost flow problem with piecewise linear convex cost functions into a minimum cost flow problem. This transformation has one significant limitation: it substantially expands the underlying network.

Recall that we are assuming that each piecewise linear convex function contains at most p linear segments. To simplify our notation, we assume that each cost function has exactly p linear segments. We incur no loss of generality in imposing this assumption because we can always add "trivial" segments of zero length at the end of the last interval $0 \le x_{ij} \le u_{ij}$. Moreover, we need not store these trivial segments explicitly because $d_{ij}^{k-1} = d_{ij}^{k} = u_{ij}$ for any such segment.

Consider a flow x_{ij} on arc (i, j), which we will view as decomposed into different segments, each representing flow between two of the breakpoints d_{ij}^{k-1} to d_{ij}^{k}. Let y_{ij}^{k} denote the flow along the kth segment, that is, between d_{ij}^{k-1} and d_{ij}^{k}. For example, if we send 6 units of flow along the arc with the cost function depicted in Figure 14.1(a), we send 2 units of flow along segment 1, 3 units along segment 2, and 1 unit along segment 3. In general, we can compute the segment flows y_{ij}^{k} from the total arc flow x_{ij} using the following formula:

$$y_{ij}^{k} = \begin{cases} 0 & \text{if } x_{ij} \le d_{ij}^{k-1} \\ x_{ij} - d_{ij}^{k-1} & \text{if } d_{ij}^{k-1} \le x_{ij} \le d_{ij}^{k} \\ d_{ij}^{k} - d_{ij}^{k-1} & \text{if } x_{ij} \ge d_{ij}^{k}. \end{cases}$$

By definition, $x_{ij} = \sum_{k=1}^{p} y_{ij}^{k}$ and $C_{ij}(x_{ij}) = \sum_{k=1}^{p} c_{ij}^{k} y_{ij}^{k}$. Substituting $\sum_{k=1}^{p} y_{ij}^{k}$ for x_{ij} in (14.1) gives us the following problem:

$$\text{Minimize} \quad z = \sum_{(i,j) \in A} \sum_{k=1}^{p} c_{ij}^{k} y_{ij}^{k} \tag{14.4a}$$

subject to

$$\sum_{\{j:(i,j) \in A\}} \sum_{k=1}^{p} y_{ij}^{k} - \sum_{\{j:(j,i) \in A\}} \sum_{k=1}^{p} y_{ji}^{k} = b(i) \qquad \text{for all } i \in N, \tag{14.4b}$$

$$0 \le y_{ij}^{k} \le d_{ij}^{k} - d_{ij}^{k-1} \qquad \text{for all } (i, j) \in A, \text{ for all } k = 1, \ldots, p. \tag{14.4c}$$

It is easy to see that (14.4) is a minimum cost flow problem on an expanded network $G' = (N, A')$ with at most p nontrivial parallel arcs corresponding to each arc $(i, j) \in A$. Figure 14.5 shows the costs and capacities corresponding to any arc (i, j). Let $(i, j)^1, (i, j)^2, \ldots, (i, j)^p$ denote these arcs, and let c_{ij}^{k} and $d_{ij}^{k} - d_{ij}^{k-1}$ denote the cost and capacity of arc $(i, j)^k$.

We now establish the equivalence between the convex cost flow problem stated in (14.1) and the minimum cost flow problem stated in (14.4). Given a flow x of

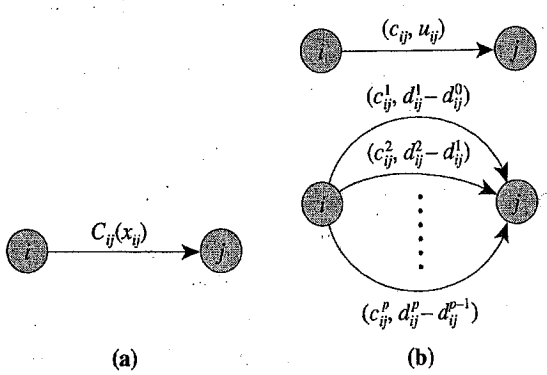


Figure 14.5 Transforming a convex cost flow problem to a minimum cost flow problem: (a) original arc; (b) corresponding arcs in the transformed network.

(14.1), we obtain the corresponding flow y of (14.4) with the same cost as follows. For each arc $(i, j) \in A$, we examine the arcs $(i, j)^1, (i, j)^2, \ldots$, in order and send the maximum possible flow along these arcs until we have sent a total of x_{ij} units from node i to node j. We refer to any such solution of (14.4) as a *contiguous solution*. The contiguous solution satisfies the property that if the flow on arc $(i, j)^k$ is positive, the flow on each of the arcs $(i, j)^1, (i, j)^2, \ldots, (i, j)^{k-1}$ equals the arc's capacity. Equivalently, in the contiguous solution, if the flow on arc $(i, j)^k$ is strictly less than its upper bound, the flow on each of the arcs $(i, j)^{k+1}, (i, j)^{k+2}, \ldots, (i, j)^p$ is zero.

Conversely, if y is a contiguous solution of (14.4), then the flow $x_{ij} = \sum_{k=1}^{p} y_{ij}^k$ is a solution of (14.1) and both the solutions have the same cost. These observations establish one-to-one correspondence between solutions of (14.1) and contiguous solutions of (14.4). A solution x_{ij} of (14.1) defined by a noncontiguous solution y_{ij}^k of (14.4) might have a different cost. But we need not worry about noncontiguous solutions, because an optimal solution of (14.4) will always be a contiguous solution. To see this, consider a noncontiguous solution in which $y_{ij}^l > 0$, and for some $k < l$, y_{ij}^k is strictly less than its upper bound. Since $c_{ij}^k < c_{ij}^l$ [by the convexity of the function $C_{ij}(x_{ij})$], we can improve this solution by adding a small number ϵ to y_{ij}^k and subtracting ϵ from y_{ij}^l. Therefore, a noncontiguous solution cannot be an optimal solution to (14.4).

The preceding discussion shows that we can solve the convex cost flow problem by solving an associated minimum cost flow problem. If y_{ij}^k is an optimal solution of the minimum cost flow problem, then $x_{ij} = \sum_{k=1}^{p} y_{ij}^k$ is an optimal solution of the convex cost flow problem.

The major drawback of the minimum cost flow transformation is that it expands the network substantially. Each arc in the convex cost flow network has as many copies in the minimum cost flow network as the number of linear segments in its cost function. When we are given the cost function $C_{ij}(x_{ij})$ specified as a piecewise linear continuous convex function for each arc $(i, j) \in A$, this transformation might be satisfactory, because the amount of storage space (i.e., input size) required to specify the convex cost flow problem is proportional to the total number of segments in all the cost functions, which equals the total number of arcs in the resulting minimum cost flow problem. In other words, both problems have the same input size. Consequently, any polynomial-time algorithm for solving the transformed minimum cost flow problem would also solve the associated convex cost flow problem in polynomial time.

On the other hand, if we were to specify the cost $C_{ij}(x_{ij})$ for some arc (i, j) as a concise function, such as x_{ij}^2, and we convert the function into piecewise linear by introducing segments of unit lengths, the transformation is not satisfactory. In this case, although stating $C_{ij}(x_{ij})$ might require only $O(1)$ space, the minimum cost flow problem will have u_{ij} copies of arc (i, j), and u_{ij} might not be polynomially bounded by n and m. Consequently, even though we employ a polynomial-time algorithm to solve the minimum cost flow problem, the resulting algorithm is not a polynomial-time algorithm for the convex cost flow problem because it is not polynomial in the problems size. To overcome this drawback, we need to develop new algorithms for the convex cost flow problem that are polynomial in the problems size. In Section 14.5 we describe one such algorithm, which we call the capacity scaling algorithm because it is a variant of the capacity scaling algorithm for the minimum cost flow

problem that we discussed in Section 10.2. As a starting point, however, we first discuss two pseudopolynomial-time algorithms.

14.4 PSEUDOPOLYNOMIAL-TIME ALGORITHMS

In this section we discuss two algorithms for the convex cost flow problem. We assume that each cost function $C_{ij}(x_{ij})$ is a piecewise linear convex function. The algorithms we discuss are modifications of the cycle-canceling algorithm and the successive shortest path algorithm discussed in Sections 9.6 and 9.7. Both algorithms use the fact that we can convert the integer version of the convex cost flow problem into a minimum cost flow problem by introducing multiple arcs. The novelty of these algorithms is that rather than introducing the multiple arcs explicitly, they handle them implicitly. These algorithms use the fact that every optimal solution of the convex cost flow problem is a contiguous solution.

Both the cycle-canceling and successive shortest path algorithms maintain a residual network at every step. Because the minimum cost flow transformation of a convex cost flow problem has multiple arcs between any pair of nodes, so does the residual network. For example, consider a flow of 3 units on an arc (i, j) of capacity 5 whose cost function is depicted in Figure 14.2. The resulting flow in the transformed network will be 1 unit on each of the arcs $(i, j)^1$, $(i, j)^2$, and $(i, j)^3$, and zero units on each of the arcs $(i, j)^4$ and $(i, j)^5$ [see Figure 14.6(b)]. Our definition of the residual network as described in Section 2.4 implies that the residual network will contain the arcs $(i, j)^4$, $(i, j)^5$, $(j, i)^1$, $(j, i)^2$, and $(j, i)^3$, each of unit capacity [see Figure 14.6(c)].

The contiguity of the solution implies that if we wish to send additional flow from node i to node j, we will send it through the arc $(i, j)^4$, and if we wish to send flow from node j to node i, we will send it through the arc $(j, i)^1$. This observation implies that we need not maintain many arcs between this pair of nodes in the residual network: Maintaining just the two arcs $(i, j)^4$ and $(j, i)^1$ is sufficient because those are the arcs that matter at this point. Eliminating multiple arcs permits us to achieve substantial savings in the storage requirements, which translates into enhanced speed of algorithms.

The preceding discussion implies the following method to construct the residual

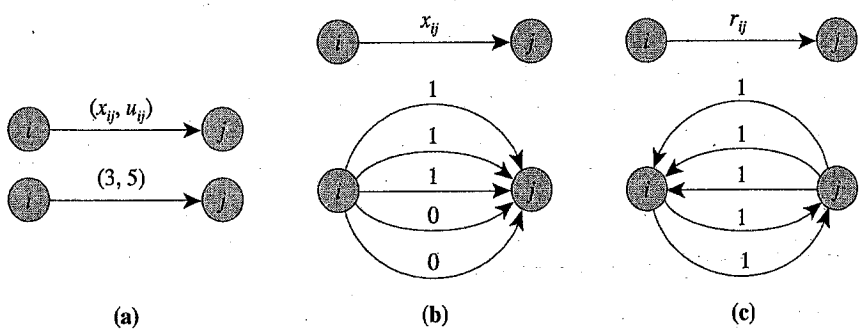


Figure 14.6 Illustrating the construction of the residual network: (a) flow on the arc (i, j) of the original network; (b) flow on arcs of the transformed network; (c) residual network for flow in the transformed network.

network $G(x)$. For any arc $(i, j) \in A$, if $x_{ij} < u_{ij}$, the residual network contains the arc (i, j) with cost $C_{ij}(x_{ij} + 1) - C_{ij}(x_{ij})$. Moreover, for any arc $(i, j) \in A$, if $x_{ij} > 0$, the residual network contains the arc (j, i) with cost $C_{ij}(x_{ij} - 1) - C_{ij}(x_{ij})$. For any arc (i, j) in the residual network, we set its residual capacity equal to the maximum flow change for which the unit flow cost remains equal to $C_{ij}(x_{ij} + 1) - C_{ij}(x_{ij})$. For instance, if the function shown in Figure 14.7 gives the cost of flow for an arc (i, j) and $x_{ij} = 7$, the residual network will contain the arc (i, j) with cost equal to 3 and residual capacity equal to 5. The residual network will also contain the arc (j, i) with cost equal to -2 and residual capacity equal to 2.

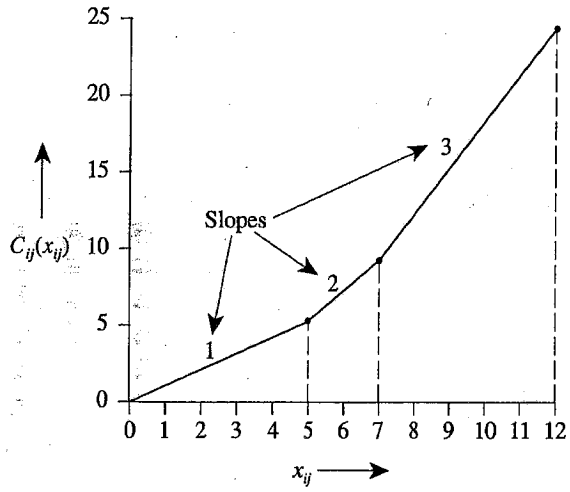


Figure 14.7 Illustrating the construction of the residual network.

We are now in a position to describe the cycle-canceling algorithm for the convex cost flow problem. We start the algorithm with a feasible flow x. We construct the residual network $G(x)$ and use any algorithm to identify a negative cycle. If the residual network does not contain any negative cycle, x is an optimal integer flow of the convex cost flow problem. If the residual network contains a negative cycle, we augment a maximum amount of flow along this cycle, update x and $G(x)$, and repeat the process. This algorithm is exactly the same as the algorithm we have discussed in Section 9.6 except that we construct the residual network differently.

To speed up the cycle-canceling algorithm in practice, we might attempt to augment more flow than would be permitted if we restrict ourselves to just one segment. At every iteration, the cycle-canceling algorithm identifies a negative cycle W and augments a maximum amount of flow along this cycle. After augmenting the flow, it updates the residual network (and the costs of arcs in the residual network) and identifies another negative cycle. At each step we might add some arcs to the residual network, delete some others, and change the costs of some arcs that remain in the residual network. However, it is quite possible that the cycle W will still be a directed cycle in the new residual network (the cost might be different, but the cycle still might be negative). If so, we augment flow along the same cycle. In fact, we can keep doing so until either W is no longer a directed cycle in the residual network or its cost becomes nonnegative. We could identify the amount of flow that we could send along W before we satisfy one of these two conditions in two ways: (1) by sending flow repeatedly along W, or (2) by performing binary search in the

interval [0, U]. We might choose the methods that would perform better in practice for the type of applications that we encounter.

The successive shortest path algorithm for the convex cost flow problem is similar. We maintain a pseudoflow x and the residual network $G(x)$ corresponding to this pseudoflow. We also maintain a set of node potentials $\pi(\cdot)$ so that the reduced cost of every arc in the residual network is nonnegative. Initially, we set each node potential $\pi(i) = 0$ and each arc flow x_{ij} equal to the value at which $C_{ij}(x_{ij})$ attains its minimum. At every iteration, the algorithm selects a node k with an excess and obtains shortest path distances $d(\cdot)$ from node k to every other node in the residual network. Then it updates the potentials by setting π to the value $\pi - d$ and augments a maximum possible flow along the shortest path from node k to some deficit node l. The algorithm repeats these steps until the pseudoflow x becomes a flow. Again, this algorithm is the same as the one that we described in Section 9.7 except that we construct the residual network differently.

This discussion shows how we can adapt two core pseudopolynomial-time minimum cost flow algorithms for the convex cost flow problem. We could also modify other algorithms for the minimum cost flow problem along similar lines. For example, in Exercise 14.20, we discuss a modification of the out-of-kilter algorithm for the convex cost flow problem, and in Exercise 14.21, we consider a modification of the network simplex algorithm.

14.5 POLYNOMIAL-TIME ALGORITHM

In this section we describe a polynomial-time algorithm for the convex cost flow problem. This algorithm is a generalization of the capacity scaling algorithm for the minimum cost flow problem discussed in Section 10.2 (this development draws heavily upon that discussion). The generalization, however, does not affect the algorithms worst-case complexity, which remains intact at $O((m \log U)S(n, m, C))$; as before, $S(n, m, C)$ is the time needed to solve a shortest path problem with nonnegative arc lengths.

The capacity scaling algorithm for the convex cost flow problem is an improvement of the successive shortest path algorithm discussed in Chapter 10. The major drawback of the successive shortest path algorithm is that it might augment as little as 1 unit of flow per augmentation, which would be *too small* compared to the total imbalances available at the nodes. As a result, the algorithm might perform *too many* augmentations. The capacity scaling algorithm ensures that the flow sent per augmentation is *sufficiently large* and as a result the total number of augmentations is *sufficiently small.*

The capacity scaling algorithm uses the following basic idea. The successive shortest path algorithm linearizes a given functional form by introducing several linear segments of unit length. The capacity scaling algorithm does not perform this linearization in a single step, but instead, does it in several scaling phases. Consider, for example, the function $C_{ij}(x_{ij}) = x_{ij}^4$ with $u_{ij} = 12$. In the first scaling phase, the algorithm linearizes the function into segments of length 8, in the second scaling phase it linearizes the function into segments of length 4, and so on until the segment lengths become 1. Figure 14.8 illustrates these linearizations for this function. The advantage of this scheme is that in the first scaling phase the algorithm can send 8

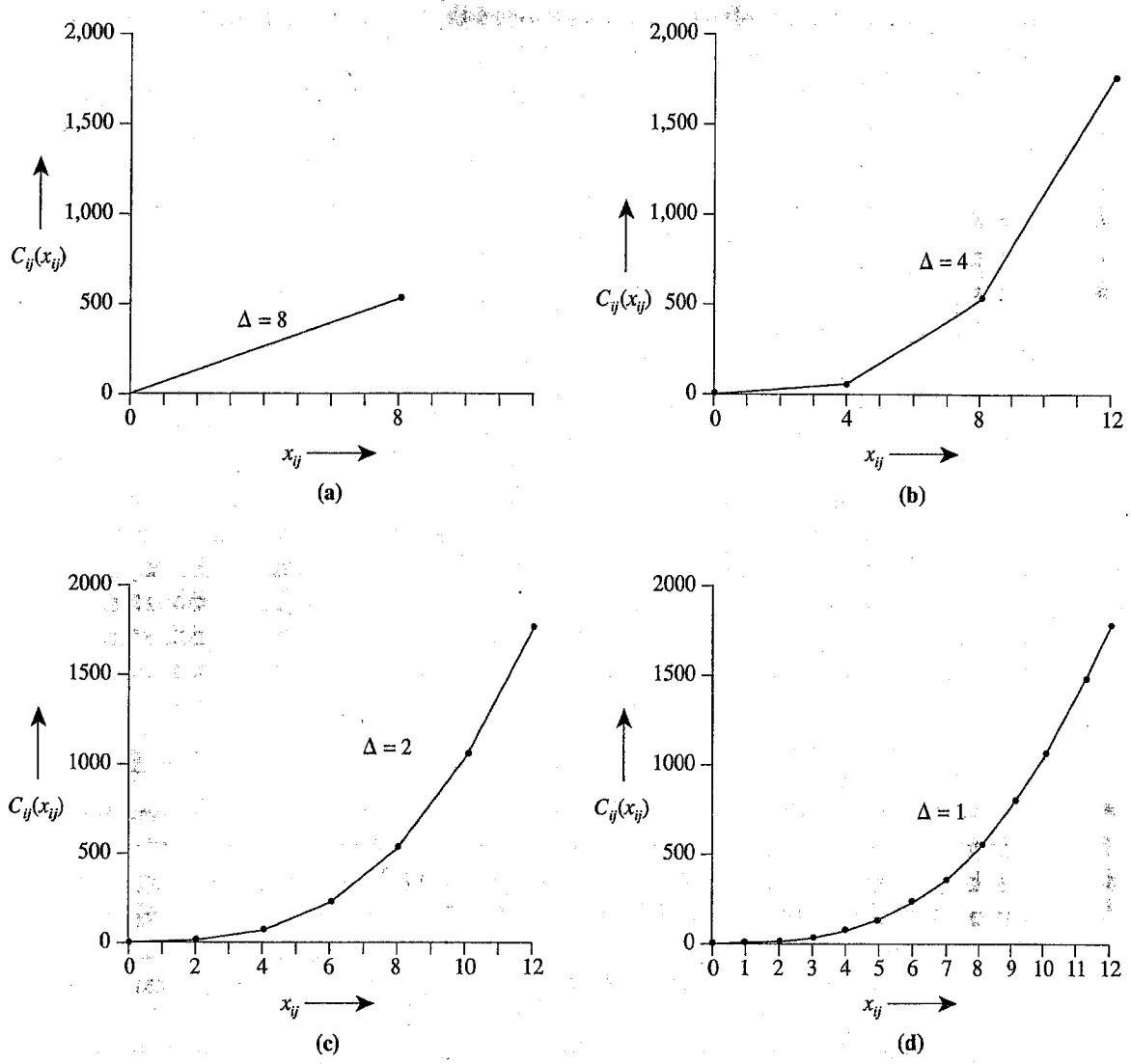


Figure 14.8 Approximations of $C_{ij}(x_{ij})$ in several scaling phases.

units of flow along augmenting paths, in the second scaling phase it can send 4 units of flow along augmenting paths, and so on. As a result, the algorithm reduces excesses at nodes at a faster rate, so it terminates more quickly.

The capacity scaling algorithm for the convex cost flow problem performs a number of scaling phases with varying values of Δ. In the Δ-scaling phase, the algorithm maintains a Δ-*residual network* $G(x, \Delta)$ with respect to a pseudoflow defined as follows: For any arc $(i, j) \in A$ with $x_{ij} + \Delta \leq u_{ij}$, the Δ-residual network contains the arc (i, j) with a residual capacity of Δ and a cost equal to $(C_{ij}(x_{ij} + \Delta) - C_{ij}(x_{ij}))/\Delta$. For any arc $(i, j) \in A$ with $x_{ij} \geq \Delta$, the Δ-residual network contains the arc (j, i) with a residual capacity of Δ and a cost equal to $(C_{ij}(x_{ij} - \Delta) - C_{ij}(x_{ij}))/\Delta$.

Initially, $\Delta = 2^{\lfloor \log U \rfloor}$ and we initiate the algorithm with the zero pseudoflow x and zero node potentials π. In the Δ-scaling phase, we first construct the Δ-residual network $G(x, \Delta)$. We then examine every arc (i, j) in A, and if any of the arcs

(i, j) or (j, i), in $G(x, \Delta)$ violates its reduced cost optimality condition (9.7), we increase or decrease the flow x_{ij} by Δ units, so that both the arcs satisfy their optimality conditions (we later show that it is always possible to do so). The algorithm next defines $S(\Delta)$ and $T(\Delta)$, respectively, as the set of nodes with excesses and deficits of at least Δ units. The algorithm then performs the following step iteratively until either $S(\Delta)$ or $T(\Delta)$ is empty: Identify a shortest path in the Δ-residual network from a node $k \in S(\Delta)$ to a node $l \in T(\Delta)$, augment Δ units of flow along this path, and update $G(x, \Delta)$. At this point the algorithm decreases Δ by a factor of 2 and starts a new scaling phase. Eventually, $\Delta = 1$ and the solution at the end of this scaling phase is optimal.

To show that the capacity scaling algorithm correctly solves the convex cost flow problem, we use the invariant property that the algorithm satisfies the reduced cost optimality condition for every arc in the Δ-residual network. Assuming that the algorithm satisfies this invariant at the beginning of each scaling phase, it is easy to show that the algorithm maintains it subsequently within that phase. The algorithm augments flow along shortest paths in the Δ-residual network, and Lemma 9.12 implies that the resulting solution satisfies the optimality conditions.

To see that the algorithm satisfies the invariant property at the beginning of the first scaling phase, notice that in this scaling phase $\Delta = 2^{\lfloor \log U \rfloor}$. This definition of Δ implies that we have linearized each cost function $C_{ij}(x_{ij})$ into at most one linear segment [as shown in Figure 14.8(a)]. If we set $x_{ij} = 0$, the Δ-residual network might contain the arc (i, j), but not (j, i); let α be the cost of arc (i, j). On the other hand, if we set $x_{ij} = \Delta$, the Δ-residual network might contain the arc (j, i) with cost $-\alpha$. We set the flow on arc (i, j) so that the cost of the corresponding arc in the Δ-residual network is nonnegative. We repeat this step for all arcs in the network G so that all arcs in the Δ-residual network have nonnegative costs. Since all node potentials are zero at this point, the reduced cost of every arc in the residual network is also nonnegative.

We next show that at the beginning of any general Δ-scaling phase, we can adjust arc flows by at most Δ units so that the Δ-residual network satisfies the reduced cost optimality conditions. We initiate the Δ-scaling phase when the 2Δ-scaling phase terminates; we assume inductively that the solution x at the end of the 2Δ-scaling phase satisfies the optimality conditions. In the 2Δ-scaling phase, we linearize $C_{ij}(x_{ij})$ by segments of length 2Δ, and in the Δ-scaling phase we linearize this cost function by segments of length Δ. Consequently, when we move from the 2Δ-scaling phase to the Δ-scaling phase, the arc costs change. As a result, the reduced costs of the arcs also change and the new values might become negative. To see this point better, consider Figure 14.9. In the 2Δ-scaling phase, the cost of the arc (i, j) is the slope of the line AB and the cost of the arc (j, i) is the negative of the slope of the line AC. In the Δ-scaling phase, the cost of the arc (i, j) is the slope of the line AD and the cost of the arc (j, i) is the negative of the slope of the line AE. We claim that by adjusting flow on an arc (i, j) by at most Δ units, we can make the reduced costs of both the arcs, (i, j) and (j, i) in the Δ-residual network nonnegative.

Suppose that x denotes the flow at the end of the 2Δ-scaling phase. At the beginning of the Δ-scaling phase, we first update the Δ-residual network and the arc reduced costs. Consider some arc (i, j) in A and assume, for simplicity, that the Δ-residual network contains both the arcs (i, j) and (j, i). The reduced costs of arcs

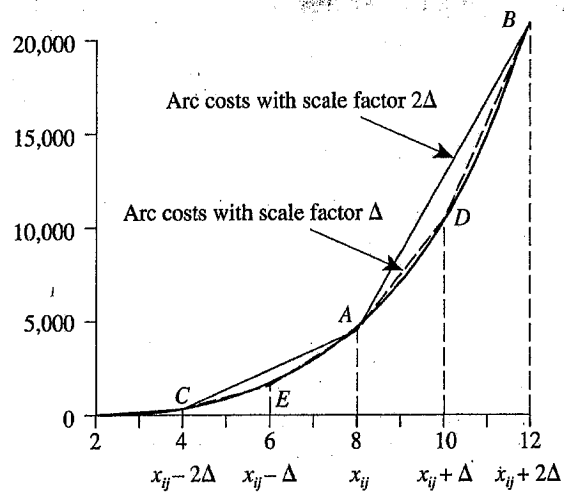


Figure 14.9 Changes in arc costs with changes in Δ.

(i, j) and (j, i) might satisfy four alternatives: (1) $c_{ij}^{\pi} \geq 0$ and $c_{ji}^{\pi} \geq 0$, (2) $c_{ij}^{\pi} < 0$ and $c_{ji}^{\pi} \geq 0$, (3) $c_{ij}^{\pi} \geq 0$ and $c_{ji}^{\pi} < 0$, and (4) $c_{ij}^{\pi} < 0$ and $c_{ji}^{\pi} < 0$. In case 1, both arcs (i, j) and (j, i) satisfy the optimality conditions and nothing needs to be done. In case 2 we increase x_{ij} by Δ units, and in case 3 we decrease x_{ij} by Δ units. Convex cost functions cannot satisfy case 4 (we ask the reader to prove this property in Exercise 14.17).

We now show that in case 2, if we increase x_{ij} by Δ units, the reduced costs of both the arcs (i, j) and (j, i) become nonnegative. The arc (i, j) in $G(x, 2\Delta)$ satisfies the reduced cost optimality condition at the end of the 2Δ-scaling phase. Therefore,

$$([C_{ij}(x_{ij} + 2\Delta) - C_{ij}(x_{ij})]/2\Delta) - \pi(i) + \pi(j) \geq 0.$$

Alternatively,

$$C_{ij}(x_{ij} + 2\Delta) - C_{ij}(x_{ij}) - 2\Delta\pi(i) + 2\Delta\pi(j) \geq 0. \tag{14.5}$$

However, the arc (i, j) in $G(x, \Delta)$ does not satisfy the reduced cost optimality condition. Consequently,

$$C_{ij}(x_{ij} + \Delta) - C_{ij}(x_{ij}) - \Delta\pi(i) + \Delta\pi(j) < 0. \tag{14.6}$$

We want to show that after we have increased x_{ij} by Δ units, the arc (i, j) in $G(x, \Delta)$ will satisfy the optimality conditions. In other words, we wish to show the following result:

$$C_{ij}(x_{ij} + 2\Delta) - C_{ij}(x_{ij} + \Delta) - \Delta\pi(i) + \Delta\pi(j) \geq 0. \tag{14.7}$$

We can write the inequality (14.5) as

$$\begin{aligned}[C_{ij}(x_{ij} + 2\Delta) - C_{ij}(x_{ij} + \Delta) - \Delta\pi(i) + \Delta\pi(j)] \\ + [C_{ij}(x_{ij} + \Delta) - C_{ij}(x_{ij}) - \Delta\pi(i) + \Delta\pi(j)] \geq 0.\end{aligned} \tag{14.8}$$

Using the expression (14.6) in (14.8) immediately implies (14.7). We also need to show that after we have increased x_{ij} by Δ units, the arc (j, i) in $G(x, \Delta)$ will also satisfy the optimality condition. To see this result, observe that the reduced cost of

arc (j, i) is $([C_{ij}(x_{ij}) - C_{ij}(x_{ij} + \Delta)]/\Delta) - \pi(j) + \pi(i)$, which is clearly positive in view of (14.6). This observation shows that in case 2, if we increase x_{ij} by Δ units, the reduced costs of both the arcs (i, j) and (j, i) are nonnegative. Using similar arguments, we can show that in case 3, if we decrease x_{ij} by Δ units, the reduced costs of both the arcs (i, j) and (j, i) are nonnegative.

To assess the complexity of the capacity scaling algorithm, note that 2Δ-scaling phase ends when either $S(2\Delta) = \emptyset$ or $T(2\Delta) = \emptyset$. Therefore, the sum of the positive imbalances is at most $2n\Delta$. At the beginning of the Δ-scaling phase, the algorithm adjusts the flow on any arc by at most Δ units. Consequently, $2(n + m)\Delta$ is a bound on the sum of the positive imbalances. Each augmentation in the Δ-scaling phase decreases the sum of the positive imbalances by Δ units; consequently, the algorithm can perform at most $O(m)$ augmentations. Overall, the algorithm performs $O(m \log U)$ augmentations and runs in $O((m \log U)S(n, m, C))$ time. We state this result as the following theorem.

Theorem 14.1. *The capacity scaling algorithm obtains an integer optimal flow for a convex cost flow problem in $O((m \log U)S(n, m, C))$ time.*

14.6 SUMMARY

The convex cost flow problem is an efficiently solvable and yet important subcase of the general nonlinear network flow problems. Unlike the minimum cost flow problem, the convex cost flow problem might not have an integer optimal solution. In this chapter, however, we assume that we wish to determine an integer optimal solution. We incur no significant loss of generality in making this assumption because by multiplying the data by a suitably large number, we can use this integer model to obtain a real-valued optimal solution that is near integer to any desired level of accuracy.

We considered two types of models: (1) each arc cost function $C_{ij}(x_{ij})$ is a piecewise linear convex function of the arc flow x_{ij}, and (2) each arc cost function is a concise convex function such as $5x_{ij}^2$. By imposing the integrality assumptions on arc flows, we can transform any concise function into a piecewise linear function by introducing unit length segments. Throughout most of our discussion, we therefore assumed that all of the arc cost functions were piecewise linear.

We first showed how to transform a convex cost flow problem with piecewise linear cost functions into a minimum cost flow problem. The major drawback of this transformation is that it expands the network substantially: For each arc of the convex cost flow network, the transformation introduces one copy of the arc into the minimum cost flow model for each linear segment in the arc's cost function. We showed that we need not maintain so many copies of each arc in the residual network $G(x)$ with respect to any flow x; maintaining at most two copies, corresponding to those arcs whose flow would change next, is sufficient. We then adapted two minimum cost flow algorithms, the cycle-canceling algorithm and the successive shortest path algorithm, for the convex cost flow problem. These algorithms are the same as those for the minimum cost flow problem with one exception: they have different residual networks. The running times of these algorithms are the same as their running times for the minimum cost flow problem.

We also discussed a polynomial-time algorithm for the convex cost flow problem. Polynomial-time algorithms for the minimum cost flow problem do not translate directly into polynomial-time algorithms for the convex cost flow problem for the simple reason that the number of arcs in the resulting minimum cost flow formulation might not be polynomially bounded in n, m, and $\log U$. As a result, we need to make modest changes in the minimum cost flow algorithms so that they retain their polynomial-time behavior. In this spirit we modified the capacity scaling algorithm for the minimum cost flow problem that we had developed in Section 10.2 to obtain a polynomial-time algorithm for the convex cost flow problem. The running time of this algorithm is $O(m \log U\, S(n, m, C))$, which is the same as that of the capacity scaling algorithm for the minimum cost flow problem. [$S(n, m, C)$ is the time required for solving a shortest path problem on a network with n nodes, m arcs, and with C as the largest arc cost.]

REFERENCE NOTES

Most of the research devoted to convex cost flows uses nonlinear programming techniques to obtain a real-valued optimal solution. In this chapter we have adopted an unconventional approach by examining methods for obtaining an integer optimal solution. The transformation of the convex cost flow problem into a minimum cost flow problem is a specialization of a standard transformation for converting a separable piecewise linear convex program into a linear program. The adaptations of the cycle-canceling and successive shortest path algorithms described in Section 14.4 are direct consequences of this transformation.

Minoux [1984] developed a polynomial-time algorithm for obtaining a real-valued optimal solution of the quadratic cost flow problem [i.e., the convex cost flow problem with arc costs of the form $C_{ij}(x_{ij}) = a_{ij}x_{ij} + b_{ij}x_{ij}^2$ for some constants a_{ij} and b_{ij}]. His approach uses the out-of-kilter algorithm as a subroutine. Subsequently, Minoux [1986] observed that this approach can also be used to obtain an integer optimal solution of the (general) convex cost flow problem. The algorithm we have presented in Section 14.5 is a variant of Minoux [1986] algorithm; it is in the framework of a scaling algorithm given by Hochbaum and Shanthikumar [1990]. Our analysis of the correctness and running time of the algorithm is similar to the analysis presented by Minoux [1986]. Goldberg and Tarjan [1987] generalized their cost scaling algorithm for the minimum cost flow problem, which we described in Section 10.3, to obtain an integer optimal solution of the convex cost flow problem if $C_{ij}(x_{ij})$ is integer for all integer x_{ij}. Hochbaum and Shanthikumar [1990] developed scaling based algorithms that would solve separable convex integer programs defined by totally unimodular constraint matrices. Their algorithm is polynomial time for finding optimal integer solutions.

Many nonlinear programming techniques are available for solving the convex cost flow problem. Among these are (1) the Frank–Wolfe method developed by Braynooghe, Gibert and Sakarovitch [1968] and Collins et al. [1978], (2) the convex simplex method described by Rosenthal [1981], (3) Newton's method as developed by Klincewicz [1983], and (4) relaxation methods proposed by Zenios and Mulvey [1986] and Bertsekas, Hosein, and Tseng [1987]. The paper by Ali, Helgason, and Kennington [1978] presents a survey of algorithms for the convex cost flow problem

developed before 1978 and the paper by Florian [1986] describes many recent developments, focusing on the use of nonlinear programming algorithms in solving transportation planning and traffic equilibrium problems.

In Section 14.2 we described several applications of the convex cost flow problem. We have adapted these applications from the following papers:

1. Urban traffic flows (Magnanti [1984]).
2. Area transfers in communication networks (Monma and Segal [1982]).
3. Matrix balancing (Schneider and Zenios [1990]).
4. Stick percolation problem (Ahlfeld, Dembo, Mulvey, and Zenios [1987]).

The model arising in electrical networks (Hu [1966]) that we described in Section 1.3 is another application of the convex cost flow problem. Some additional applications of the convex cost flow problem are (1) the target-assignment problem (Manne [1958]), (2) solution of Laplace's equation (Hu [1967]), (3) production scheduling problems (Ratliff [1978]; Barr and Turner [1981]), (4) the pipeline network analysis problem (Collins et al. [1978]), (5) microdata file merging (Barr and Turner [1981]), and (6) market equilibrium problems (Barros and Weintraub [1986]). Papers by Ali, Helgason, and Kennington [1978], Dembo, Mulvey, and Zenios [1989], and Schneider and Zenios [1990] provide additional references concerning applications of the convex cost flow problem.

EXERCISES

Note: In the following exercises, interpret an optimal solution of the convex cost flow problem as an integer optimal solution. Moreover, unless we specifically describe the form of the cost function $C_{ij}(x_{ij})$, assume that it is a piecewise linear convex function or a concise function, whichever is more convenient.

14.1. A function $f(x)$ of an n-dimensional vector x is *convex* if $f(\lambda x_1 + (1 - \lambda)x_2) \leq \lambda f(x_1) + (1 - \lambda)f(x_2)$ for every two distinct values x_1 and x_2 of x and for every weighting parameter λ, $0 \leq \lambda \leq 1$. Suppose that $f(x)$ and $g(x)$ are both convex functions of a scalar x. Which of the following functions $h(\cdot)$ are always convex functions? Justify your answer.

(a) $h(x) = f(x) + g(x)$
(b) $h(x) = f(x) - g(x)$
(c) $h(x) = (f(x))^2$
(d) $h(x) = \sqrt{f(x)}$ [Assume that $f(x) \geq 0$.]

14.2. Let x and $c^1, c^2, \ldots, c^k$ be n-dimensional vectors. Show that the function $f(x) = \max\{c^1x, c^2x, \ldots, c^kx\}$ is a convex function of x. Is the function $g(x) = \min\{c^1x, c^2x, \ldots, c^kx\}$ also convex?

14.3. Consider a minimum cost flow problem whose supply/demand vector $b(\lambda) = b^0 + \lambda b^*$ is a function of a scalar parameter λ. Let $z(\lambda)$ denote the optimal objective function value of the problem as a function of this parameter. Show that $z(\lambda)$ is a convex function of λ.

14.4. Capacity expansion of a network. A network $G = (N, A)$ is used to send flow from one node s to another node t and does not have sufficient arc capacities to meet anticipated future demands. Suppose that we wish to increase some of the arc capacities so that we can send the desired amount of flow from node s to node t. Let α_{ij} denote the per unit cost of increasing the capacity of arc (i, j). Suppose that we wish to determine an

expansion plan that increases the maximum flow in the network to v^0 while incurring the least possible cost. Formulate this problem as a convex cost flow problem.

14.5. Finding nearly feasible flows. Recall from Section 9.7 that a pseudoflow x of a network flow problem is a solution that satisfies the arc flow bounds $0 \le x_{ij} \le u_{ij}$, but might violate the mass balance constraints. To determine whether the network flow problem has a feasible flow, and if not, then to determine a pseudoflow with minimum possible infeasibility, we could attempt to find a pseudoflow x that minimizes the function $\sum_{i \in N} [e(i)]^2$ of the excesses

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

(Observe that the minimum value of this problem is zero if and only if the network flow problem has a feasible solution.) Show how to formulate this excess minimization problem as a convex cost flow problem. (*Hint*: Augment the network by adding a node and some arcs.)

14.6. Racial balancing with penalties. Consider the racial balancing problem described in Application 9.3. Assume that each school j has a targeted enrollment of $\overline{b}_j$ black students and $\overline{w}_j$ white students. The actual number of black and white students enrolled in the jth school might differ from these targeted values. Suppose that if we miss any of these targets by y students, we incur an associated penalty of $|\alpha_j|y$. We would like to allocate students to the schools in a way that minimizes the sum of the total cost of transportation and the penalties. Formulate this problem as a convex flow problem.

14.7. Solve the convex cost flow problem shown in Figure 14.10(a) by the cycle-canceling algorithm. Assume that arc (i, j) has the flow cost $c_{ij}x_{ij}^2$ for the value of c_{ij} specified in the figure. Start with the following flow: $x_{13} = x_{34} = 5$, and $x_{ij} = 0$ for all other arcs. Always augment flow along a negative cycle with the minimum cost. Show the residual network after each augmentation.

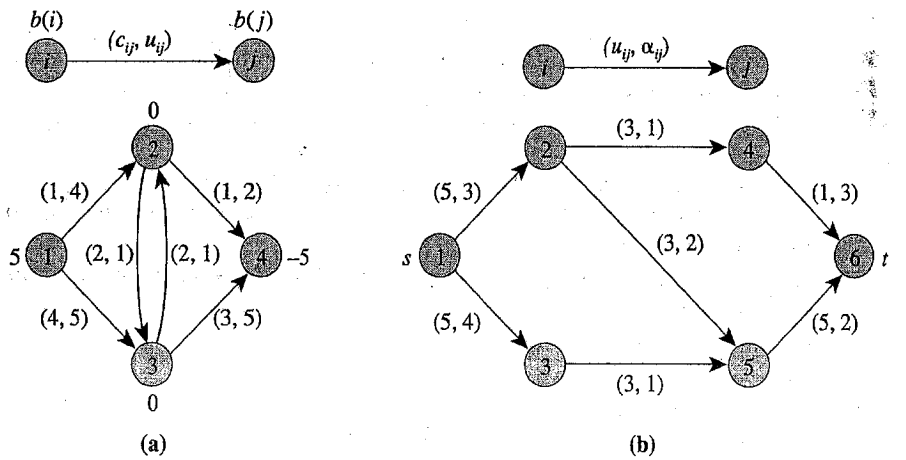


Figure 14.10 Examples for Exercises 14.7 to 14.9.

14.8. This exercise concerns the capacity expansion problem described in Exercise 14.4. Consider a numerical example of this capacity expansion problem shown in Figure 14.10(b) and assume that we wish to send 10 units of flow from the source node to the sink node. Solve this problem by the successive shortest path algorithm.

14.9. Apply the capacity scaling algorithm to the convex cost flow problem shown in Figure 14.10(a). Show your computations for only the first two scaling phases.

14.10. In a particular class of convex cost flow problems formulated on *undirected* networks, the flow cost on any arc (i, j) satisfies the conditions $C_{ij}(x_{ij}) \ge 0$ for all $x_{ij} \ge 0$ and

$C_{ij}(0) = 0$. Show how to convert this type of problem into a convex cost flow problem on a directed network. Justify your transformation by establishing an equivalence between the flows in the two networks. What happens if we relax the assumption that $C_{ij}(0) = 0$?

14.11. Let x^* be an optimal solution of a convex cost flow problem with $C_{ij}(x_{ij})$ as the flow cost on arc $(i, j) \in A$. Suppose that we add a constant k to each arc cost; that is, $C'_{ij}(x_{ij}) = C_{ij}(x_{ij}) + k$ for some constant k. Professor May B. Wright claims that x^* also solves the modified problem. Prove or disprove this claim.

14.12. In Section 11.2 we proved that a minimum cost flow problem always has at least one optimal spanning tree solution [i.e., a solution with an associated spanning tree that satisfies the condition that each nontree arc (i, j) has a flow value of $x_{ij} = 0$ or of $x_{ij} = u_{ij}$ and each tree arc (i, j) has flow x_{ij} satisfying the flow bounds $0 \le x_{ij} \le u_{ij}$]. Show that convex cost flow problems do not satisfy this spanning tree property. To do so, construct an instance of the convex cost flow problem (with piecewise linear convex functions or with concise convex functions) that has a unique nonspanning tree solution.

14.13. We say that a function $C_{ij}(x_{ij})$ is *concave* if for any two points x'_{ij} and x''_{ij} and for every value of the parameter θ satisfying $0 \le \theta \le 1$, $C_{ij}(\theta x'_{ij}) + (1 - \theta)x''_{ij}) \ge \theta\, C_{ij}(x'_{ij} + (1 - \theta)C_{ij}(x''_{ij})$. Consider a capacitated network flow problem in which the cost of flow $C_{ij}(x_{ij})$ on each arc (i, j) is a concave function. In this network we want to obtain a flow that minimizes the total cost of flow. Show that this problem always has an optimal spanning tree solution. Explain why this result is not true for the convex cost flow problem. (*Hint*: Use an approach similar to the one we used in Section 11.2 to show that we can always obtain a cycle free solution from any given solution without increasing the cost of the flow.)

14.14. Let x^* be an optimal solution of the convex cost flow problem. Describe an $O(m)$ method that either shows that x^* is the unique solution to the problem or that finds an alternative optimal solution.

14.15. Let x^* be a feasible solution of the convex cost flow problem. Consider the residual network $G(x^*)$ as defined in Section 14.4. Show that x^* is an optimal solution of the convex cost flow problem if and only if $G(x^*)$ contains no negative cycle.

14.16. In Section 14.4, while describing the cycle-canceling algorithm for the convex cost flow problem, we indicated that we can use a binary search technique to determine the maximum flow δ that we can augment along the selected cycle W so that it remains a negative cycle. Work out the details of this method and specify the time needed to determine δ.

14.17. While discussing the capacity scaling algorithm for the convex cost flow problem in Section 14.5, we claimed that at the beginning of any Δ-scaling phase, we will never encounter a pair of arcs (i, j) and (j, i) in the residual network satisfying the conditions $c^{\pi}_{ij} < 0$ and $c^{\pi}_{ji} < 0$. Prove this claim.

14.18. Budget-constrained capacity expansion. In this exercise we study a variation of the capacity expansion problem described in Exercise 14.4. Suppose that we have allocated D dollars for increasing arc capacities (assume that D is integer). We wish to spend this money in a way that will permit the maximum possible flow from node s to node t in the network. Suggest a polynomial-time algorithm for solving this problem. (*Hint*: Formulate the problem as a constrained maximum flow problem as in Exercise 10.25 and use the solution technique developed in that exercise.)

14.19. Consider the budget-constrained capacity expansion problem described in Exercise 14.18. Show how to solve this problem by a single application of the parametric network simplex algorithm described in Section 11.9.

14.20. Suppose that we wish to develop a generalization of the out-of-kilter algorithm (see Section 9.9) for solving a convex cost flow problem whose arc costs are all piecewise linear convex functions. Specify a kilter diagram for an arc (i.e., those combinations of reduced costs c^{π}_{ij} and arc flows x_{ij} that satisfy the optimality conditions). Next define

the kilter number of an arc (i, j) as the flow change required to make it an in-kilter arc. Finally, show that by solving a shortest path problem, we can reduce the kilter number of some arc by at least 1 unit.

14.21. Adapt the network simplex algorithm for a convex cost flow problem whose arc costs are each given by a concise function $C_{ij}(x_{ij})$. Explain how to perform the following steps: (1) identifying an entering (k, l); (2) determining the maximum flow that we can augment along the cycle formed by adding arc (k, l) to the spanning tree; and (3) updating the node potentials.

14.22. Cost scaling algorithm. In this exercise we discuss an adaptation of the cost scaling algorithm for the minimum cost flow problem discussed in Section 10.3 for solving a convex cost flow problem when each arc cost is a piecewise linear convex function containing at most p linear segments. Suppose that we transform this problem into a minimum cost flow problem and then use the generic version of the cost scaling algorithm described in Section 10.3. For a specific scaling phase, obtain a bound on the number of times that the algorithm performs each of the following operations: (1) saturating pushes; (2) relabels; and (3) nonsaturating pushes. How much time does the algorithm require to execute a scaling phase? What is the running time of the entire algorithm?

14.23. Suppose that we wish to obtain a real-valued optimal flow for a convex cost flow problem whose arc cost functions $C_{ij}(x_{ij})$ are all concise functions. Let x be a feasible flow for the convex cost flow problem and let ϵ be any positive real number. Let $G(x)$ denote the residual network with respect to the flow x. We define the *ϵ-incremental costs* of arcs in the residual network in the following manner. If $(i, j) \in A$ and $x_{ij} < u_{ij}$, then $G(x)$ contains the arc (i, j) with an ϵ-incremental cost equal to $[C_{ij}(x_{ij} + \epsilon) - C_{ij}(x_{ij})]/\epsilon$. If $(i, j) \in A$ and $x_{ij} > 0$, then $G(x)$ contains the arc (i, j) with the ϵ-incremental cost equal to $[C_{ij}(x_{ij} - \epsilon) - C_{ij}(x_{ij})]/\epsilon$. Show that x is a real-valued optimal solution of the convex cost flow problem if and only if for all $\epsilon > 0$, $G(x)$ contains no directed cycle with a negative ϵ-incremental cost. Use this result to outline an algorithm that produces a real-valued optimal solution of the problem to any desired degree of accuracy (i.e., produces a solution whose objective function value is sufficiently close to the optimal objective function value).

15

GENERALIZED FLOWS

There are occasions when it is undoubtedly better to incur loss than to make gain.
—Titus Maccius Plautus

Chapter Outline

15.1 INTRODUCTION

In each of the models we have considered so far, we have made one very fundamental, yet almost invisible, assumption: We conserve flow on every arc. That is, the amount of flow on any arc that leaves its tail node equals the amount of flow that arrives at its head node. This assumption is very reasonable in many application settings, including the numerous applications we have considered in the previous chapters (and that we consider later in Chapter 19). Other practical contexts, however, violate this conservation assumption. For example, in the transmission of a volatile gas, we might lose flow because of evaporation; or, in the transmission of liquids such as raw petroleum crude, we might lose flow due to leakage.

In this chapter we consider a basic *generalized network flow model* for addressing these situations. In this model we associate a positive multiplier μ_{ij} with every arc (i, j) of the network and assume that if we send 1 unit from node i to node j along the arc (i, j), then μ_{ij} units arrive at node j. This model is a generalization of the minimum cost flow problem that we have been considering in previous chapters in the sense that if every multiplier has value 1, the generalized network flow model becomes the minimum cost flow problem.

The generalized maximum flow problem is another special case of the generalized network flow problem. In this model, instead of determining a minimum cost flow, we determine a maximum flow that can leave the source or that can enter the sink. The literature on the generalized maximum flow problem is extensive and includes several recently developed polynomial-time algorithms. In this chapter, rather than discussing these algorithms, we concentrate on the generalized minimum

cost flow problem because it is more general and includes the generalized maximum flow problem as a special case. Moreover, rather than attempting to be comprehensive in our coverage of algorithms for the generalized minimum cost flow problem, we will study just one algorithm, an adaptation of the network simplex method, which we refer to as the *generalized network simplex algorithm*.

Like our discussion of the network simplex method in Chapter 11, our presentation of the generalized network simplex algorithm emphasizes the problem's underlying combinatorial structure. We do, however, require limited background in linear programming since a linear programming perspective simplifies much of our development. The combinatorial and linear programming approaches both provide valuable insight into the generalized network simplex method. We stress the combinatorial approach because it is similar to the way that we have developed the network simplex method in Chapter 11 and because it requires only modest background in linear programming.

Recall from Chapter 11 that the network simplex algorithm maintains a partitioning of the arcs of the network as a triple (**T**, **L**, **U**) called a spanning tree structure. The arcs in **T** correspond to those in a spanning tree and the arcs in **U** and **L** are nontree arcs with flow at their upper and lower bounds. The rationale for restricting our search to this type of solution rests on the fundamental spanning tree property that implies that any minimum cost flow problem always has a spanning tree solution. The generalized network simplex algorithm will be conceptually similar. We again restrict our attention to a particular type of solution (**F**, **L**, **U**), called an *augmented forest structure*; in this case, the arcs in **F** constitute what we call an *augmented forest*. As in the network simplex method, the generalized algorithm will be an iterative procedure, moving from one augmented forest structure to another, at each step producing an augmented forest structure with a smaller cost (assuming nondegeneracy). To guide the algorithmic steps, we again define node potentials and use them to determine optimality conditions for assessing when a given solution is optimal.

We have organized this chapter in a modular fashion. After describing a number of applications of the generalized flow problem, we begin to develop the generalized network simplex algorithm by defining augmented forests and by describing several of their properties, including optimality conditions for assessing when an augmented forest structure defines an optimal solution. We then develop the two major building blocks of the generalized network simplex algorithm: procedures for finding the node potentials and the arc flows associated with any augmented forest structure. Not coincidentally, these procedures are also the major building blocks of the simplex method for general linear programs. To highlight the connection between our development in this chapter and linear programming in general, we next show that an augmented forest is a graph-theoretic interpretation of a linear programming basis of the linear programming formulation of the generalized network flow problem. In Section 15.6 we bring all of these algorithm ingredients together to produce the generalized network simplex algorithm.

To set notation, let us first introduce the following linear programming formulation of the generalized flow problem:

$$\text{Minimize} \quad \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{15.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} \mu_{ji}x_{ji} = b(i) \qquad \text{for all } i \in N, \tag{15.1b}$$

$$0 \le x_{ij} \le u_{ij} \qquad \text{for all } (i, j) \in A. \tag{15.1c}$$

As we have already noted, $\mu_{ij} > 0$ is the *multiplier* of the arc (i, j). We assume that each arc multiplier μ_{ij} is a rational number, that is, it can be expressed as $\mu_{ij} = p_{ij}/q_{ij}$ for some integers p_{ij} and q_{ij}. When we send 1 unit of flow on arc (i, j), μ_{ij} units of flow arrive at node j. If $\mu_{ij} < 1$, the arc is *lossy*; if $\mu_{ij} > 1$, the arc is *gainy*.

Notice that we are assuming that the arc capacity u_{ij} is an upper bound on the flow that we send from node i, not on the flow that becomes available at node j. Similarly, c_{ij} is the cost per unit flow that we send from node i, not the per unit cost of the flow that becomes available at node j. In this model we assume that the lower bound on every arc flow is zero. Exercise 15.9 shows that we incur no loss of generality in making this assumption.

15.2 APPLICATIONS

Generalized networks can successfully model many application settings that cannot adequately be represented as minimum cost flow problems. Two common interpretations of the arc multipliers underlie many uses of generalized flows. In the first interpretation, we view the arc multipliers as modifying the amount of flow of some particular item. Using this interpretation, generalized networks model situations involving physical transformations such as evaporation, seepage, deterioration, and purification processes with various efficiencies, as well as administrative transformations such as monetary growth due to interest rates. In the second interpretation, we view the multiplication process as transforming one type of item into another. This interpretation allows us to model processes such as manufacturing, currency exchanges, and the translation of human resources into job requirements. In the following discussion, we describe applications of the generalized network flows that use one or both of these interpretations of the arc multipliers.

Application 15.1 Conversions of Physical Entities

In many different application settings, generalized networks arise quite naturally because the flow in a network converts one type of physical entity into another at a certain conversion rate. The following few brief problem descriptions are illustrative of these types of applications.

Financial networks. In financial networks, nodes represent various equities such as stocks, bonds, current deposits, Treasury bills, and certificates of deposit at certain points in time and arcs represent various investment alternatives that convert one type of equity into another. The multiplier of an arc represents the gain associated with the corresponding investment.

Mineral networks. In these networks, nodes represent mines, purification plants, refineries, ports, and final markets. Arcs represent processing opportunities or flow of material through intermediate junctions to their final destinations. The multiplier of an arc represents the loss associated with the corresponding process.

Energy networks. As discussed in Application 1.9, in certain types of energy networks, nodes represent various raw materials (e.g., crude oil, coal, uranium, or hydropower) and various energy outputs (e.g., electricity, domestic oil, or gas). The arcs represent the transformation of one raw material into an energy output; the efficiency of this transformation is the arc multiplier.

Application 15.2 Machine Loading

Machine loading problems arise in a variety of application domains. In one of the most popular contexts, we would like to schedule the production of r products on p machines. Suppose that machine i is available for α_i hours and that any of the p machines can produce each product. Producing 1 unit of product i on machine j consumes a_{ij} hours of the machine's time and costs c_{ij} dollars. To meet the demands of the products, we must produce β_j units of product j. In the machine loading problem, we wish to determine how we should produce, at the least possible production cost, the r products on the p machines.

In this problem setting, products compete with each other for the use of the more efficient, faster machines; the limited availability of these machines forces us to use the less economical and slower machines to process some of the products. To achieve the optimal allocation of products to the machines, we can formulate the problem as a generalized network flow problem, as shown by Figure 15.1. The network has p product nodes, $1, 2, \ldots, p$ and r machine nodes, $\bar{1}, \bar{2}, \ldots, \bar{r}$. Product node i is connected to every machine node $\bar{j}$. The multiplier a_{ij} on arc $(i, \bar{j})$ indicates the hours of machine capacity needed to produce 1 unit of product i on machine j. The cost of the arc $(i, \bar{j})$ is c_{ij}. The network also has arcs $(\bar{j}, \bar{j})$ for each machine

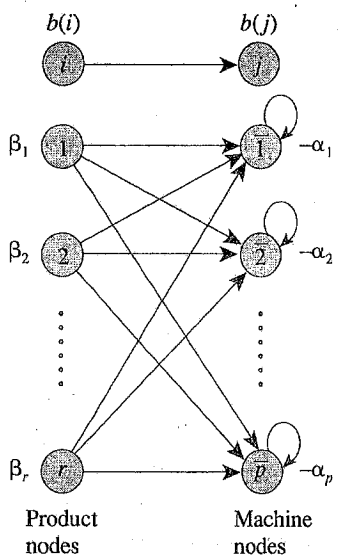


Figure 15.1 Formulating a machine loading problem as a generalized network flow problem.

node $\bar{j}$; the multiplier of each of these arcs is 2 and the cost is zero. The purpose of these arcs is to account for the unfulfilled capacities of the machines: as described in Section 15.3, we can send additional flow along these arcs to generate enough flow (or items) to exactly consume the available machine capacity.

As stated, this machine loading problem is fairly generic. Consequently, it arises naturally in many different problem settings. The following example illustrates one such application context. We leave the detailed formulation of this problem as an exercise (see Exercise 15.3).

Aircraft assignment. An airline needs to assign its fleet of various aircrafts to its flight routes. The airline fleet has β_i aircraft of type i; it wishes to use this fleet to meet its demand of α_j passengers on each of its j routes. By operating an aircraft of type i on route j, the airline incurs a cost of c_{ij} and can accommodate a_{ij} passengers. The airline would like to assign aircrafts to the routes to satisfy the customer demand at the least possible operating costs. Note that the generalized network flow formulation of the aircraft assignment problem does not assure that the number of aircrafts assigned to a route will be integral. The optimal solution might be fractional. In some cases, rounding up this fractional solution to an integer solution might provide a good solution of the problem. In other cases, the solution to the generalized flow problem would serve as a good starting point, and as a valuable bounding mechanism, for initiating an implicit enumeration procedure.

Application 15.3 Managing Warehousing Goods and Funds Flows

An entrepreneur owns a warehouse of fixed capacity H that she uses to store a price-volatile product. Knowing the price of this product over the next K time periods, she needs to manage her purchases, sales, and storage patterns. Suppose that she holds I_0 units of the good and C_0 dollars as her initial assets. In each period she can either buy more goods or sell the goods in the warehouse to generate additional cash. The price of the product varies from period to period and ultimately all goods must be sold. The problem is to identify a buy–sell strategy that maximizes the amount of cash C_K available at the end of the Kth period.

Figure 15.2 gives a generalized network flow formulation of this warehousing

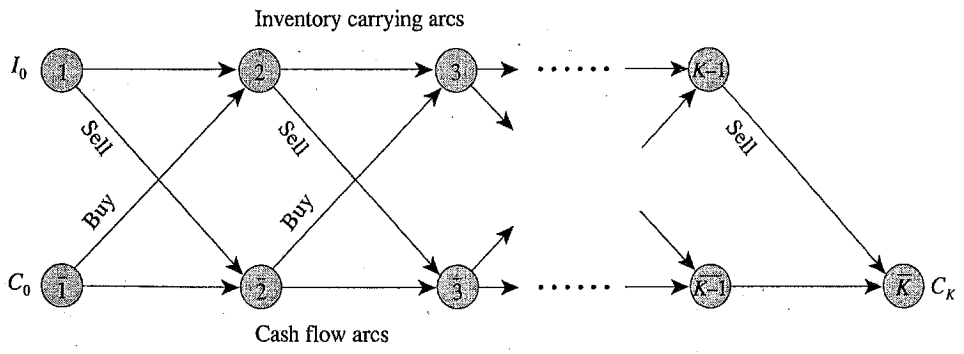


Figure 15.2 Formulating the warehouse funds flow model.

goods and funds flow problem. This formulation has the following three types of arcs:

1. *Inventory carrying arcs*. The cost of this type of arc is the inventory carrying cost per unit; its capacity is H. The multiplier on this arc is less than 1 or equal to 1, depending on whether carrying inventory incurs any loss.
2. *Cash flow arcs*. These arcs are uncapacitated and have zero cost. The multiplier of this type of arc is the bank interest rate for that period.
3. *Buy and sell arcs*. These arcs also are uncapacitated and have zero cost. If p_i is the purchase cost of the product in period i, a buy arc has a multiplier of value $1/p_i$ and a sell arc has a multiplier of value p_i.

It is easy to establish a one-to-one correspondence between buy–sell strategies of the entrepreneur and flows in the underlying network. To enrich this model, we could easily incorporate various additional features, such as policy limits on the maximum and minimum amount of cash held or goods flow in each period or introduce alternative investments such as certificates of deposits that extend beyond a single period.

Application 15.4 Land Management

The U.S. Bureau of Land Management (BLM) manages 173 million acres of public rangelands. It uses a significant part of this land to grow vegetation consumed by animals (both wild and domestic). The BLM must devise a resource management plan for determining the optimal number of animals of different types that the land can support, given the vegetation inventory and the dietary requirements for the different animal types. We present a simplified version of this problem.

The BLM needs to support several animal types, say $A_1, A_2, \ldots, A_a$, using several types of vegetation, say $V_1, V_2, \ldots, V_v$, while satisfying the following constraints:

1. The total animal consumption cannot exceed an upper limit β_j (in pounds) on the production of vegetation V_j. This upper limit prescribes the maximum amount of the annual vegetation production that the animals can remove by grazing without reducing the vigor of this type of vegetation. The Bureau uses historical records and professional judgment to determine these limits.
2. Each animal type i consumes α_i units of vegetation. Animal type A_i can consume at most γ_{ij} pounds of vegetation type V_j. This bound defines a fraction of the total annual vegetation production that a given animal type can consume without destroying the surrounding vegetation community (different animals might have a different effect on the vegetation).
3. Each animal type must receive a "balanced diet" that will satisfy certain dietary requirements. These requirements are stated as follows: The ratio of the intake of vegetation of type V_j (in pounds) to the total intake of all vegetation for each animal type A_i must lie between f_{ij} and g_{ij}. The Bureau determines these limits from the scientific literature.

Figure 15.3 shows a generalized network flow model for a situation with two animal types and three vegetation types, but restricted to only the first two of the three listed constraints. In the figure any arc other than the source arc has a multiplier of value 1. The flow on a source arc (s, i) indicates the number of animals of type A_i supported. Since the multiplier of this arc is α_i, when this flow reaches node i, it is converted into the total food requirement of the animal type A_i. The network distributes this food requirement among different vegetation types while honoring the imposed lower and upper limits. If we set the cost of each source arc to be -1, this model will determine the maximum number of animal types that the land can support.

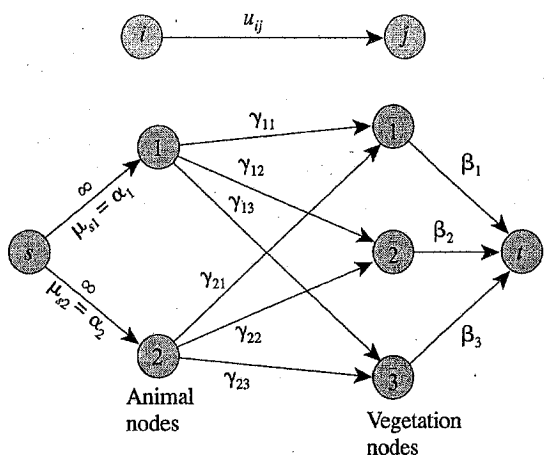


Figure 15.3 Generalized network flow formulation of the land management problem.

Incorporating the third constraint in the model formulation adds additional complications. This constraint states that $f_{ij}(\alpha_i x_{si}) \leq x_{ij} \leq g_{ij}(\alpha_i x_{si})$ for each animal type A_i and vegetation type V_j. Unfortunately, these constraints destroy the network structure of the model. However, we can still use network flow techniques to solve the problem because it has significant embedded network flow structure. By using a technique known as the Lagrangian relaxation (which we discuss in Chapter 16), we can relax—that is, remove—these "nonnetwork" constraints and solve the original problem by repeatedly solving a sequence of generalized network flow problems.

15.3 AUGMENTED FOREST STRUCTURES

In this section we present background material needed for developing the generalized network simplex algorithm for the generalized network flow problem. This algorithm maintains a topological structure that we call an *augmented forest structure*. In this section we define the augmented forest structure and derive associated optimality conditions for it. In Section 15.5 we show that an augmented forest structure defines a linear programming basis structure for the generalized network flow problem.

Flows along Paths

Let P be a path (not necessarily, directed) from node s to node t. Let $\overline{P}$ and $\underline{P}$ denote the sets of forward and backward arcs in P. We define the *path multiplier* $\mu(P)$ of the path P as follows:

$$\mu(P) = \frac{\Pi_{(i,j)\in\overline{P}}\mu_{ij}}{\Pi_{(i,j)\in\underline{P}}\mu_{ij}} \tag{15.2}$$

We first address the following question: If we send a unit amount of flow from node s to node t along P, how does the arc flow change? Consider, for example, the path shown in Figure 15.4(a); suppose that we wish to send 2 units of flow from node 1 to node 5. To send 2 units from node 1 to node 5 means that 2 units leave node 1, a certain amount, say α, reaches node 5, and inflow equals outflow at all the internal nodes of the path. If we send 2 units on the arc (1, 2), 6 units become available at node 2 because the multiplier of this arc is 3. The arc (2, 3) has multiplier 0.5, so when we send 6 units on it, only 3 units reach node 3. If we carry the flow further, then 12 units reach node 4 as well as node 5. To summarize, if we send 2 units along the path 1–2–3–4–5, then 12 units reach node 5. The ratio of units reaching node 5 to the units sent from node 1 is 12/2 = 6, which equals the multiplier of the path.

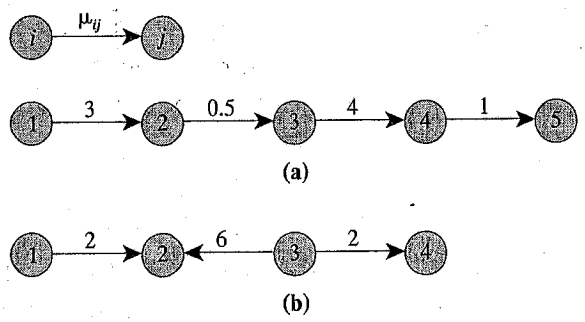


Figure 15.4 Flows along paths in a generalized network: (a) path with all forward arcs, path multiplier is 6; (b) path with both forward and backward arcs, path multiplier is $\frac{2}{3}$.

Next, suppose that we send 1 unit along the path 1–2–3–4 shown in Figure 15.4(b). If we send 1 unit on the arc (1, 2), 2 units become available at node 2. The next arc on the path, arc (3, 2), is a backward arc. We need to send enough flow on this arc to cancel the 2 units at node 2. If we send $-\frac{1}{3}$ of a unit on the arc (3, 2), -2 units become available at node 2, thus satisfying the mass balance constraint at node 2. But sending $-\frac{1}{3}$ of a unit on arc (3, 2) creates an excess of $\frac{1}{3}$ of a unit at node 3. We next send $\frac{1}{3}$ of a unit on arc (3, 4) and $\frac{2}{3}$ of a unit becomes available at node 4. We thus find that if we send 1 unit along the path 1–2–3–4, then $\frac{2}{3}$ of a unit reaches node 4, which again equals the multiplier of the path.

As illustrated by these two examples, (1) if we send y units of flow on a forward arc (i, j), the flow creates $\mu_{ij}y$ units at node j; and (2) if we send y units from node j on a backward arc (j, i), the flow on the arc is $-y/\mu_{ji}$ units and y/μ_{ji} units of flow become available at node i. The following property is an immediate consequence of the preceding discussion.

Property 15.1. *If we send* 1 *unit of flow from node s to another node t along a path P, then* $\mu(P)$ *units become available at node t.*

Flows along Cycles

Let W be a cycle (not necessarily directed) from a specified node s to itself whose orientation has already been defined. Let $\overline{W}$ and $\underline{W}$ denote the sets of forward and backward arcs in this cycle. With respect to the cycle's orientation, we define its cycle multiplier $\mu(W)$ as follows:

$$\mu(W) = \frac{\Pi_{(i,j)\in\overline{W}}\mu_{ij}}{\Pi_{(i,j)\in\underline{W}}\mu_{ij}}. \tag{15.3}$$

Sending flow along a cycle is the same as sending flow along a path except that the flow comes back to itself. Property 15.1 implies that if we send 1 unit of flow along the cycle W starting from node s, then $\mu(W)$ units return to this node. If $\mu(W) > 1$, we create an excess at node s; in this case we refer to the cycle W as a *gainy cycle*. If $\mu(W) < 1$, we create a deficit at node s; in this case we refer to the cycle W as a *lossy cycle*. If $\mu(W) = 1$, the flow around this cycle conserves mass balance at all its nodes; we refer to any such cycle W as a *breakeven cycle*.

Notice that if we reverse the orientation of the cycle, we exchange the roles of the sets $\overline{W}$ and $\underline{W}$ and as a result, the numerator in the expression (15.3) becomes the denominator, and the denominator becomes the numerator. The following result formalizes this observation:

Property 15.2. *If* $\mu(W)$ *is the multiplier of a cycle W with a particular orientation, then* $1/\mu(W)$ *is the multiplier of the same cycle with the opposite orientation.*

Note that unless the cycle is a breakeven cycle, we can make it either a gainy cycle or a lossy cycle by defining its orientation appropriately. We next state some additional properties of cycles that are immediate consequences of the preceding discussion. The proofs of these properties are straightforward and left to the reader.

Property 15.3. *By sending (i.e., augmenting)* θ *units along a nonbreakeven cycle W starting at node s, we create an imbalance of* $\theta(\mu(W) - 1)$ *units at node s.*

Property 15.4. *Let s be a node in a nonbreakeven cycle W. Then to uniquely create an imbalance of* α *units at node s (while satisfying the mass balance constraints at all other nodes), we must send* $\alpha/(\mu(W) - 1)$ *flow along the cycle W starting at node s.*

Augmented Tree and Augmented Forest

Property 15.4 implies that in a feasible solution of the generalized network flow problem, the set of arcs A' with positive flow will not, in general, be a spanning tree. To ensure feasibility, the arcs in A' might contain a cycle. For example, if the network itself is a cycle W with a gain $\mu(W)$, one node t has a positive demand and each node other than node t has a zero demand, the problem has a unique solution

and the set A' of arcs with positive flow will be the entire cycle. Therefore, for the generalized network flow problem, each component of A' might contain a cycle. As we show later in this chapter, the generalized network flow problem always has an optimal solution for which each component of A' contains exactly one cycle (assuming nondegeneracy). These types of solutions play the same central role in generalized flows that spanning tree solutions play in minimum cost flows. In this section we describe these special types of solutions and develop optimality conditions for them.

Let $G^a = (N^a, T^a)$ be a subgraph of $G = (N, A)$ so that $N^a \subseteq N$ and $T^a \subseteq A$. We refer to G^a as an *augmented tree* if T^a is a spanning tree of the node set N^a together with an additional arc (α, β) which we call the *extra arc*. An augmented tree has a specially designated node, called its *root*. We consider any augmented tree as hanging from its root. Figure 15.5(b) and (c) show two augmented trees of the graph shown in Figure 15.5(a). In these figures we depict the extra arcs by dashed lines.

An augmented tree contains exactly one cycle which is formed by adding the extra arc (α, β) to the tree $T^a - \{(\alpha, \beta)\}$; we refer to this cycle as the *extra cycle*. Note that we can consider any arc in the extra cycle as the extra arc. For reasons that will become clear later, we refer to an augmented tree as a *good augmented tree* if its extra cycle is lossy or gainy (i.e., not a breakeven cycle).

We define an *augmented forest* $G^f = (N, F)$ with $F \subseteq A$ as a collection of node-disjoint augmented trees that span all the nodes of the graph. We refer to an augmented forest as a *good augmented forest* if each of its components is a good augmented tree. Figure 15.5(d) shows an augmented forest of the graph shown in Figure 15.5(a). We refer to those arcs in an augmented forest as the *augmented-forest arcs* and the remaining arcs as *nonaugmented-forest arcs*.

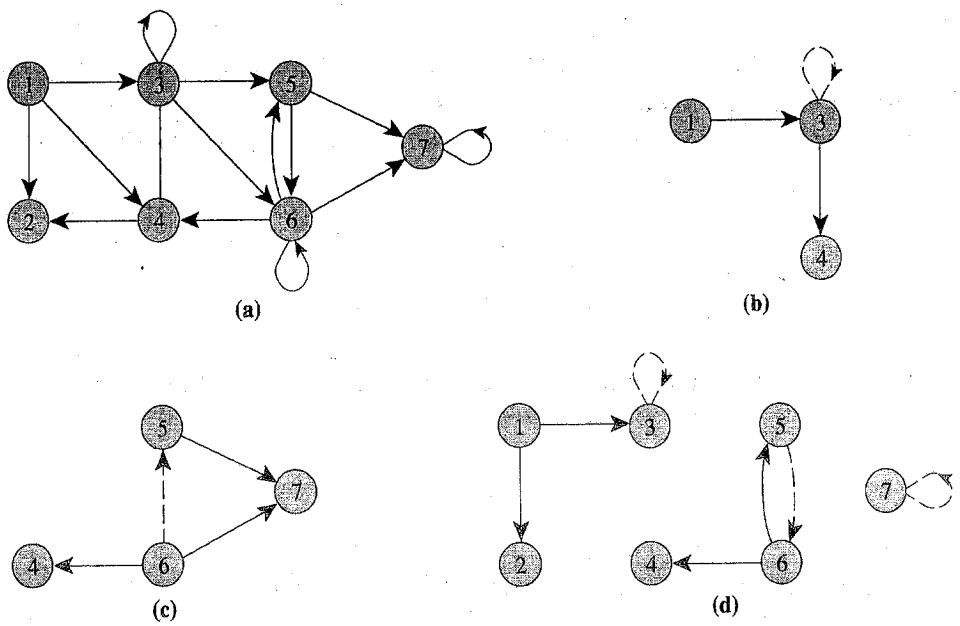


Figure 15.5 Examples of augmented trees and augmented forest: (a) original graph; (b) and (c) two augmented trees; (d) augmented forest.

We store an augmented forest in a computer as a collection of augmented trees. Each augmented tree is a tree plus the extra arc. We can store a tree by associating three indices with each node i in the tree: a predecessor index, $pred(i)$, a depth index, $depth(i)$, and a thread index, $thread(i)$. We refer the reader to Section 11.3 for a detailed discussion of these indices. These indices allow us to perform operations on trees very efficiently.

Augmented Forest Structures and Optimality Conditions

Suppose that the sets **F**, **L**, and **U** define a partition of the arc set A and that **F** is a good augmented forest. As before, we refer to the arcs in **F** as AF-arcs. We refer to the arcs in **L** as nonaugmented-forest arcs at their lower bounds, and the arcs in **U** as nonaugmented-forest arcs at their upper bounds. We also refer to the triple (**F**, **L**, **U**) as an *augmented forest structure*.

An augmented forest structure (**F**, **L**, **U**) is either feasible or infeasible. If we set $x_{ij} = 0$ for all $(i, j) \in \mathbf{L}$ and $x_{ij} = u_{ij}$ for all $(i, j) \in \mathbf{U}$, then a unique flow on the arcs of the augmented forest will satisfy the system of equations (15.1b) (in Section 15.4 we show how to compute this flow). If this flow satisfies the lower and upper bound constraints imposed on all the arcs of the augmented forest, we say that the structure (**F**, **L**, **U**) is *feasible*; otherwise, it is *infeasible*. We will say that a feasible augmented forest structure is *nondegenerate* if $0 < x_{ij} < u_{ij}$ for every arc $(i, j) \in \mathbf{F}$; it is *degenerate* otherwise. We will also say that a feasible augmented forest structure (**F**, **L**, **U**) is an *optimal augmented forest structure* if its associated flow x_{ij} is an optimal solution of (15.1).

We associate with each node i a number $\pi(i)$, which we refer to as its *node potential*. With respect to a set of node potentials, we define the *reduced cost* of an arc (i, j) as $c_{ij}^{\pi} = c_{ij} - \pi(i) + \mu_{ij}\pi(j)$. In the following theorem, we state and prove a sufficiency condition for a flow to be optimal.

Theorem 15.5 (Generalized Flow Optimality Conditions). *A flow x^* is an optimal solution of the generalized network flow problem if it is feasible and for some vector π of node potentials, the pair (x^*, π) satisfies the following optimality conditions:*

$$(a) \quad \text{If } 0 < x_{ij}^* < u_{ij}, \text{ then } c_{ij}^{\pi} = 0. \tag{15.4a}$$

$$(b) \quad \text{If } x_{ij}^* = 0, \text{ then } \quad c_{ij}^{\pi} \geq 0. \tag{15.4b}$$

$$(c) \quad \text{If } x_{ij}^* = u_{ij}, \text{ then } \quad c_{ij}^{\pi} \leq 0. \tag{15.4c}$$

Proof. We first claim 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 proof of this claim is similar to that of Property 2.4 and is left as an exercise (see Exercise 15.12). Let π be a vector that together with the flow x^* satisfies the conditions (15.4), and let x be any arbitrary flow. Consider the following summation:

$$\sum_{(i,j)\in A} c_{ij}^{\pi}(x_{ij} - x_{ij}^*). \tag{15.5}$$

We claim that each term in (15.5) is nonnegative. We establish this claim by considering three cases.

Case 1: $0 < x_{ij}^* < u_{ij}$. In this case (15.4a) implies that $c_{ij}^\pi = 0$, so the term $c_{ij}^\pi(x_{ij} - x_{ij}^*)$ is zero.

Case 2: $x_{ij}^* = 0$. In this case $x_{ij} \geq x_{ij}^* = 0$, and by (15.4b), $c_{ij}^\pi \geq 0$, so the term $c_{ij}^\pi(x_{ij} - x_{ij}^*)$ is nonnegative.

Case 3: $x_{ij}^* = u_{ij}$. In this case, $x_{ij} \leq x_{ij}^* = u_{ij}$, and by (15.4c), $c_{ij}^\pi \leq 0$, so the term $c_{ij}^\pi(x_{ij} - x_{ij}^*)$ is again nonnegative.

We have shown that $c^\pi(x - x^*) = c^\pi x - c^\pi x^* \geq 0$, or $c^\pi x^* \leq c^\pi x$, which concludes the proof of the theorem. ◆

The following property is an immediate consequence of this optimality condition.

Property 15.6 (Augmented Forest Structure Optimality Conditions). *A feasible augmented forest structure* **(F, L, U)** *with the associated flow* x^* *is an optimal augmented forest structure if for some vector* π *of node potentials, the pair* (x^*, π) *satisfies the following optimality conditions:*

(a) $c_{ij}^\pi = 0$ *for all* $(i, j) \in \mathbf{F}$. (15.6a)

(b) $c_{ij}^\pi \geq 0$ *for all* $(i, j) \in \mathbf{L}$. (15.6b)

(c) $c_{ij}^\pi \leq 0$ *for all* $(i, j) \in \mathbf{U}$. (15.6c)

15.4 DETERMINING POTENTIALS AND FLOWS FOR AN AUGMENTED FOREST STRUCTURE

Associated with each augmented forest structure are unique arc flows and a unique set of node potentials; in this section we describe efficient methods for determining these quantities. These methods are the major subroutines of the generalized network simplex algorithm that we describe in Section 15.6. We begin by considering the computation of node potentials.

Determining Node Potentials for an Augmented Forest Structure

Let (F, L, U) be an augmented forest structure of the generalized network flow problem. The augmented forest F contains several augmented trees. We describe a method for determining node potentials for the nodes in an augmented tree. Applying this method iteratively for every augmented tree, we can obtain potentials for every node in the network. Let $T \cup \{(\alpha, \beta)\}$ be the augmented tree under consideration and let node h be its root. We wish to determine node potentials that satisfy the condition $c_{ij}^\pi = 0$ for every arc (i, j) in the augmented tree $T \cup \{(\alpha, \beta)\}$. We first set the potential of node h equal to a parameter θ whose numerical value we will compute later. We then fan out along the tree arcs (i, j) using the thread indices and compute the other node potentials by using the equation $c_{ij}^\pi = c_{ij} - \pi(i) + \mu_{ij}\pi(j) = 0$. The thread traversal ensures (see Section 11.3) that we have already evaluated one of the potentials, $\pi(i)$ or $\pi(j)$, so we can compute the other from the equation $c_{ij} -$

$\pi(i) + \mu_{ij}\pi(j) = 0$. We note that all the node potentials determined in this way will be (linear) functions of θ. We next use the equation for the extra arc, $c_{\alpha\beta} - \pi(\alpha) + \mu_{\alpha\beta}\pi(\beta) = 0$, to compute a numerical value of θ. This numerical value of θ allows us to compute the numerical values of all the node potentials. Figure 15.6 gives an algorithmic description of this method.

```
procedure compute-potentials;
begin
    π(h) : = θ;
    j : = thread(h);
    while j ≠ h do
    begin
        i : = pred(j);
        if (i, j) ∈ A then π(j) : = (π(i) − c_ij)/μ_ij;
        if (j, i) ∈ A then π(j) : = μ_ji π(i) + c_ji;
        j : = thread(i)
    end;
    for each node i, let the potential π(i), as a function of θ, be represented
            by f(i) + g(i)θ for some constants f(i) and g(i);
    compute θ : = (c_αβ − f(α) + μ_αβ f(β)) / (g(α) − μ_αβ g(β));
    substitute this value of θ in the expression π(i) = f(i) + θg(i) to
            compute the potentials for each node i;
end;
```

Figure 15.6 Computing node potentials of an augmented tree.

Let us illustrate this procedure on a numerical example. Figure 15.7(a) shows an augmented tree and Figure 15.7(b) shows the node potentials in terms of the parameter θ. Using the extra arc (2, 3), we compute $\theta = -17$. Substituting for this value of θ permits us to determine the numerical values of all node potentials, as shown in Figure 15.7(c). We suggest that the reader verify that all the arcs in the augmented tree have zero reduced costs.

The correctness of this procedure follows from the definition of the node potentials [i.e., the vector π must satisfy the condition $c_{ij}^{\pi} = 0$ for every arc (i, j) in the augmented tree $T \cup \{(\alpha, \beta)\}$]. To show that we can carry out the steps of these computations, we need to show that while computing the numerical value of θ in the procedure *compute-potentials* (see Figure 15.6), the denominator $(g(\alpha) - \mu_{\alpha\beta}g(\beta))$ is never zero. We ask the reader to establish this result in Exercise 15.20. The procedure *compute-potentials* has a complexity of $O(n)$.

Determining Flow for an Augmented Forest Structure

Let **(F, L, U)** be an augmented forest structure of the generalized network flow problem. Assume for the time being that the network is uncapacitated, and consequently, $\mathbf{U} = \emptyset$. As before, we describe a method for determining the flow for an augmented tree; applying this method for every augmenting tree individually, we can determine flows on all the arcs of the augmented forest. The method for computing the arc flows proceeds in the reverse fashion of the method we have used for computing the node potentials: Instead of starting at the root node and fanning out along the tree arcs, we start at the leaf nodes and move in toward the root.

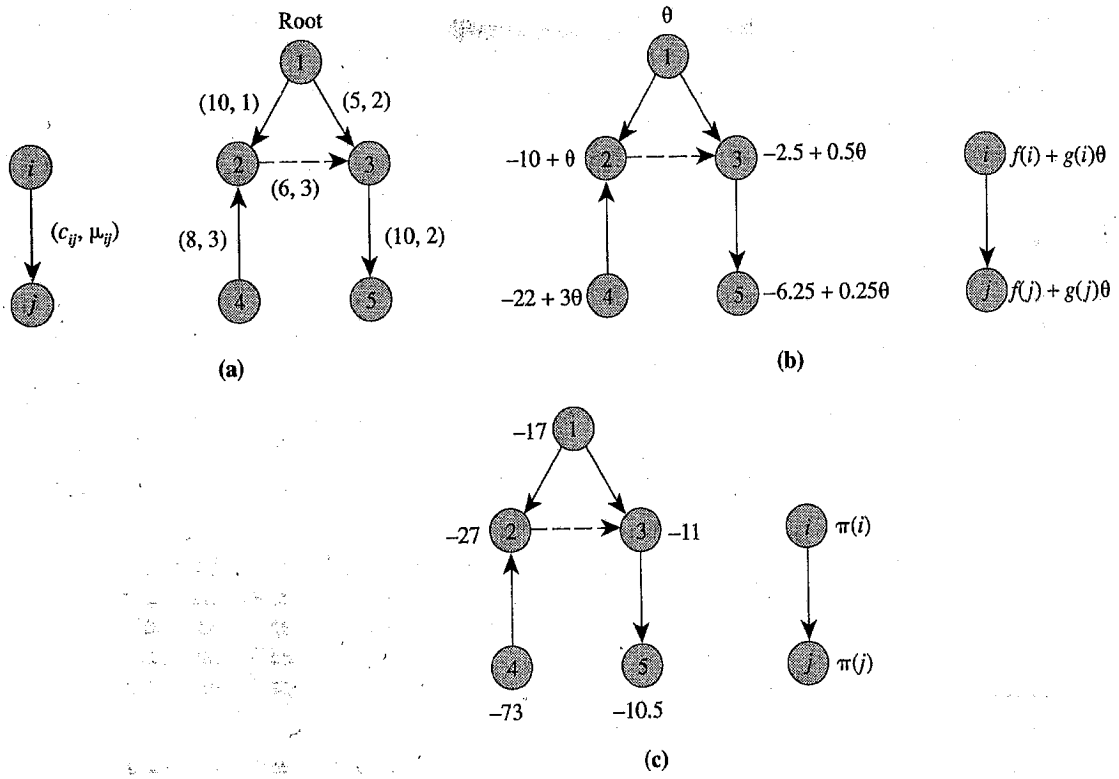


Figure 15.7 Illustrating the computation of node potentials.

Our method for determining the flow for an augmenting tree $T \cup \{(\alpha, \beta)\}$ with root h works as follows. We first define the *imbalance* $e(i)$ of each node i as equal to $b(i)$ and set the flow on each arc equal to zero. We then set the flow on the extra arc (α, β) equal to θ. This amounts to decreasing the imbalance at node α by θ units and increasing the imbalance at node β by $\mu_{\alpha\beta}\,\theta$ units. We next determine all of the arc flows as a function of θ; then we determine the numerical value of θ, and substituting this value for the arc flows, we determine the numerical values of all the arc flows.

Let us consider a leaf node j in the tree T. Exactly one tree arc (i.e., an arc in T) is incident to node j: It is either (j, i) or (i, j). Therefore, we have only one way to discharge the imbalance of node j, through the arc (j, i) or (i, j). If arc (j, i) is incident to node j, we send $e(j)$ units of flow from node j to node i, which reduces the imbalance of node j to zero, sets the flow on arc (j, i) equal to $e(j)$, and changes the imbalance of node i to $e(i) + e(j)\mu_{ji}$ [because $e(j)\mu_{ji}$ additional units of flow arrive at node i]. We illustrate this case in Figure 15.8(a). If arc (i, j) is incident to node j, we need to send $-e(j)/\mu_{ij}$ units of flow from node i over the arc (i, j) in order to make $-e(j)$ units available at node j, thus canceling its excess. We illustrate this possibility in Figure 15.8(b). In this case we change the imbalance of node i to $e(i) - e(j)/\mu_{ij}$ and set the flow on arc (i, j) to $-e(j)/\mu_{ij}$. Once we have determined the flow value on the arc (i, j) or (j, i), whichever is present in the network, we delete this arc and repeat the procedure on the remaining tree. Eventually, we are left with only the root node h with an imbalance $e(h)$.

At this point, the flow on each arc in the augmented tree $T \cup \{(\alpha, \beta)\}$ is a linear

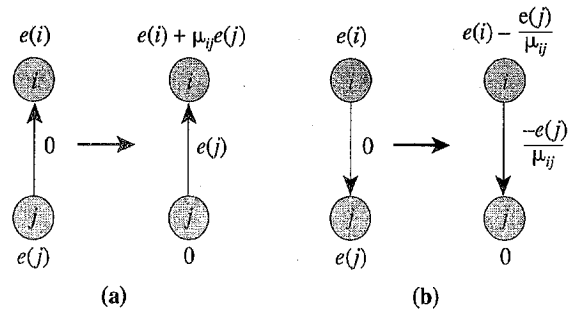


Figure 15.8 Computing flows on tree arcs.

function of θ and the flow satisfies the mass balance constraints of all the nodes except node h, which has an imbalance of $e(h)$ (which again is a linear function of θ). By setting $e(h) = 0$, we compute θ and determine the numerical values of flows on the arcs in the augmented tree. Figure 15.9 summarizes our discussion in the form of an algorithmic description of the procedure. We apply the procedure shown in Figure 15.9 for every augmented tree, starting from the following global initialization: $e(i) = b(i)$ for all $i \in N$ and $x_{ij} = 0$ for all $(i, j) \in A$.

```
procedure compute-flows;
begin
    set x_αβ := θ, e(α) := e(α) − θ, and e(β) := e(β) + μ_αβ θ;
    T' := T;
    while T' ≠ {h} do
    begin
        select a leaf node j in T';
        i := pred(j);
        if (j, i) ∈ T' then set x_ji := e(j) and add −e(j)μ_ji to e(i);
        if (i, j) ∈ T' then set x_ij := −e(j)/μ_ij and add e(j)/μ_ij to e(i);
        delete node j and the arc incident to it from T';
    end;
    for each arc (i, j) in the augmented tree, let x_ij be represented by f(i, j) + θg(i, j);
    let e(h) be represented by f(h) + θg(h);
    compute θ := −f(h)/g(h) and use this value of θ to compute numerical values of all the arc
      flows;
end;
```

Figure 15.9 Determining arc flows for an augmented tree.

It is easy to verify that this procedure uniquely determines the flow on arcs of the augmented tree since by scanning tree arcs we determine the arc flows uniquely as a function of θ, and the mass balance constraints of the root node h imply a unique numerical value of θ.

We illustrate this procedure using the numerical example shown in Figure 15.10(a). The figure shows the node imbalances after we have sent an amount of flow θ on the extra arc (2, 3). We examine nodes of the trees in the order 4, 5, 2, 3, and compute the flows on the arcs incident to these nodes. Figure 15.10(b) shows the arc flows and node imbalances at this point. Now, node 1 has an imbalance of $25 - 0.5\theta$, and equating this quantity to zero, we find that $\theta = 50$. Using this value of θ, we compute the numerical values for all the arc flows, as shown in Figure 15.10(c).

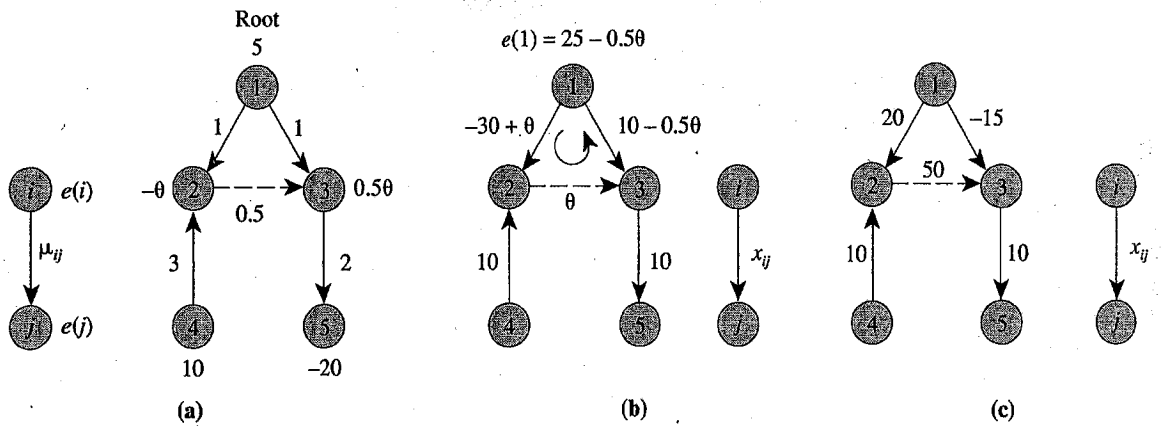


Figure 15.10 Illustrating the computation of flow for an augmented tree.

To complete our discussion showing that the procedure *compute-flows* correctly finds the arc flows, we need to show that $g(h)$ is never zero; otherwise, we cannot compute θ using the equation $\theta = -f(h)/g(h)$. It is easy to see that $\theta g(h)$ is the imbalance at node h resulting from setting the flow on the extra arc (α, β) to value θ. Setting the flow on the arc (α, β) to value θ creates a deficit of $-\theta$ units at node α and an excess of $\mu_{\alpha\beta}\theta$ units at node β. Let P_α denote the tree path from node h to node α and P_β denote the tree path from node β to node h; also, let $\mu(P_\alpha)$ denote the multiplier of the path P_α. We can cancel the deficit at node α by sending $\theta/\mu(P_\alpha)$ units from node h to node α. Similarly, when we send the excess of $\mu_{\alpha\beta}$ units from node β to node h on the path P_β, $\mu_{\alpha\beta}\theta\mu(P_\beta)$ units arrive at node h. These observations show that

$$g(h) = \mu_{\alpha\beta}\mu(P_\beta) - 1/\mu(P_\alpha).$$

Therefore, $g(h)$ is zero if and only if $\mu(P_\alpha)\mu_{\alpha\beta}\mu(P_\beta) = 1$. Since $\mu(P_\alpha)\mu_{\alpha\beta}\mu(P_\beta)$ is the multiplier of the extra cycle, we have shown that $g(h)$ is zero if and only if the extra cycle is a breakeven cycle. Since the augmented tree is good, the extra cycle is not a breakeven cycle and $g(h)$ is nonzero.

So far we have considered uncapacitated networks, that is, we have assumed that $\mu_{ij} = \infty$ for all $(i, j) \in A$. As a result, the set **U** is empty. If the network is capacitated and the set **U** is nonempty, we need to slightly modify the procedure *compute-flows*. In the uncapacitated case, we start with $x_{ij} = 0$ for all $(i, j) \in A$ and $e(i) = b(i)$ for all $i \in N$. In the capacitated case we start with the same values of x_{ij} and $e(i)$ and then execute the following statements:

```
for every (i, j) ∈ U do
begin
    xij := uij;
    e(i) := e(i) − uij;
    e(j) := e(j) + μijuij;
end;
```

The purpose of these statements is to set the flow on each arc $(i, j) \in \mathbf{U}$ to its upper bound, which creates an additional deficit of u_{ij} units at node i and an additional

excess of $\mu_{ij}u_{ij}$ units at node j. After invoking this initialization, we execute the procedure *compute-flows* as described in Figure 15.9.

The procedure *compute-flows* provides us with a method for determining a flow that satisfies the supply/demand constraints of all the nodes in an augmented tree. Applying this procedure repeatedly for each augmented tree, we obtain a flow that satisfies the supply/demand constraints of all the nodes. Note that we have computed the arc flows by applying the procedure *update-flows* and using the fact that the flow satisfies the mass balance constraints. We have observed earlier that the flows on the arcs in the augmented forest that satisfy the mass balance constraints are unique and the procedure *compute-flows* determines this unique solution. This flow might or might not satisfy the flow bounds on the arc flows. If it does, the augmented forest structure is *feasible*; otherwise, it is *infeasible*. Clearly, the running time of the procedure *compute-flows* is $O(m)$.

15.5 GOOD AUGMENTED FORESTS AND LINEAR PROGRAMMING BASES

In this section we establish a connection between the good augmented forests of G and bases of the generalized network flow problem. The graph-theoretic structure of the basis allows us to specialize the linear programming simplex method so that we can perform all of the computations on the network itself. As a result, the generalized network simplex algorithm is substantially faster than the general-purpose simplex method. Our main result in this section uses a well-known property of linear programs.

In stating this result, we consider a linear program formulated as

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

subject to

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

$$0 \leq x \leq u.$$

In this formulation, $\mathcal{A}$ is a $p \times q$ matrix whose rows are linearly independent (i.e., the matrix has rank equal to p). We let $\mathcal{A}_i$ denote the column in $\mathcal{A}$ associated with the variable x_i, let $\mathbf{B}$ denote an index set of p variables, $x_{\mathbf{B}} = \{x_i : i \in \mathbf{B}\}$, and $\mathcal{B} = \{\mathcal{A}_i : i \in \mathbf{B}\}$. We use the following well-known result:

Property 15.7. *The variables $x_{\mathbf{B}}$ define a basis of the linear programming problem if and only if the system of equations $\mathcal{B}x_{\mathbf{B}} = b$ has a unique solution.*

When translated into the framework of the generalized network flow problem, this property implies that a subset $\mathbf{B}$ of arcs defines a basis of the generalized network flow problem if and only if for every supply/demand vector b, the arcs in $\mathbf{B}$ have a unique flow (not necessarily honoring the flow bounds) that satisfies the mass balance constraints. We will show that arcs in $\mathbf{B}$ have a unique flow if and only if $\mathbf{B}$ is a good augmented forest.

Recall from the last section that if $\mathbf{B}$ is a good augmented forest, the flow on each augmented forest arc is unique. Therefore each augmented forest defines a

basis of the generalized network flow problem. We next establish the converse result: If **B** is not a good augmented forest, it cannot be a basis. To do so, we use another well-known result stating that each basis of a linear program contains the same number of variables. Since each good augmented forest contains n arcs and defines a basis, each basis **B** of the generalized network flow problem must contain n arcs. Now consider a set **B** of n arcs that does not define a good augmented forest. Let us consider those components (i.e., connected subgraphs) of **B** that are not good augmented trees. Since some component is not a good augmented tree, either some component is a spanning tree (Case 1), or some component is an augmented tree whose extra cycle is breakeven (Case 2). (As a third possibility, some component might have two or more extra arcs; but notice that in this case some other component must satisfy Case 1; therefore, it is sufficient to consider Case 1 and Case 2.) We consider these two cases separately.

Case 1.

Let $\hat{\mathbf{B}}$ be a component of **B** that is a tree. Designate an arbitrary node, say node h, as the root of $\hat{\mathbf{B}}$. Set $b(h) = 1$ and $b(i) = 0$, for each node $i \in N - \{h\}$. As is easy to verify, we can never consume the supply at node h if we restrict the flow to only tree arcs. This conclusion violates the basis property that for every vector b, some flow must satisfy the mass balance constraints.

Case 2.

Let $\hat{\mathbf{B}}$ be a component of **B** that is an augmented tree whose extra cycle W is breakeven. Notice that sending additional flow around a breakeven cycle maintains the mass balance constraints at all the nodes of the cycle. Therefore, if $\hat{\mathbf{B}}$ has a feasible flow, it has infinitely many feasible flows. This conclusion violates the basis property that for every vector b, a unique flow satisfies the mass balance constraints.

The preceding observations establish the following result.

Theorem 15.8. *A set* **B** *of arcs defines a basis of the generalized network flow problem if and only if* **B** *is a good augmented forest.*

Since each good augmented forest constitutes a basis of the generalized network flow problem, each good augmented forest structure defines a basis structure of the generalized network flow problem.

15.6 GENERALIZED NETWORK SIMPLEX ALGORITHM

The generalized network simplex algorithm is similar to the network simplex algorithm that we discussed in Chapter 11. The algorithm maintains a (good) feasible augmented forest structure at every iteration (which, in linear programming terminology, is a feasible basis structure) and by performing a pivot operation transforms this solution into an improved (good) augmented forest structure. The algorithm repeats this process until the augmented forest structure satisfies the optimality conditions (15.6). Figure 15.11 gives an algorithmic description of the generalized network simplex algorithm.

```
algorithm generalized network simplex;
begin
    determine an initial feasible augmented forest structure (F, L, U);
    let x be the flow and π be the node potentials associated
        with the initial augmented forest structure;
    while some nonaugmented forest arc violates its optimality condition do
    begin
        select an entering arc (k, l) violating its optimality condition;
        add arc (k, l) to the augmented forest and determine the leaving arc (p, q);
        update the the solutions x and π and the augmented forest structure;
    end;
end;
```

Figure 15.11 Generalized network simplex algorithm.

In the following discussion we describe in greater detail various steps of this algorithm.

Obtaining an Initial Augmented Forest Structure

It is easy to obtain an initial (all-artificial) augmented forest structure. For every node $i \in N$ we first introduce an artificial arc (i, i) of sufficiently large cost M and infinite capacity. We set the multiplier of the arc (i, i) equal to 0.5 if node i is a supply node [i.e., $b(i) > 0$], and equal to 2 if node i is a demand or a transshipment node [i.e., $b(i) \leq 0$]. Notice that for a supply node i, because the cycle consisting of the arc (i, i) is a lossy cycle, we can consume the supply of node i by sending flow along this cycle. Similarly, for a demand node i, the cycle (i, i) is a gainy cycle and by augmenting flow along the cycle we can generate sufficient flow to satisfy the demand of node i. Property 15.4 implies that by setting $x_{ii} = e(i)/(1 - \mu_{ii})$, we can satisfy the supply/demand of the node i. Moreover, notice that since each artificial arc (i, i) has sufficiently large cost M, no solution with a positive flow on any artificial arc will be optimal unless the generalized network flow problem is infeasible. We determine the initial node potentials from the fact that the reduced cost of each arc in $\mathbf{F}$ must be zero. Using $c_{ii}^{\pi} = c_{ii} - \pi(i) + \mu_{ii}\pi(i)$ for each node $i \in N$ yields $\pi(i) = M/(1 - \mu_{ii})$.

Optimality Testing and Entering Arc

Let $(\mathbf{F}, \mathbf{L}, \mathbf{U})$ denote a feasible augmented forest structure of the generalized network flow problem and let π be the corresponding node potentials. To determine whether the augmented forest structure $(\mathbf{F}, \mathbf{L}, \mathbf{U})$ is optimal, we check to see whether it satisfies the following optimality conditions:

$$c_{ij}^{\pi} \geq 0 \qquad \text{for every arc } (i, j) \in \mathbf{L},$$

$$c_{ij}^{\pi} \leq 0 \qquad \text{for every arc } (i, j) \in \mathbf{U}.$$

If the current augmented forest structure satisfies these conditions, it is optimal and the algorithm terminates. Otherwise, the algorithm selects a nonaugmented forest arc violating its optimality condition and introduces this arc into the augmented forest. Two types of arcs are *eligible* to enter the augmented forest:

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 generalized network simplex algorithm can select any eligible arc as the entering arc. However, different rules, known as *pivot rules*, for selecting the entering arc produce algorithms with different empirical behavior. In Chapter 11 we discussed several popular pivot rules for the network simplex algorithm. These were (1) *Dantzig's pivot rule*, which selects the arc with maximum violation as the entering arc; (2) the *first eligible arc pivot rule*, which selects, in a wraparound fashion, the first arc with positive violation encountered in examining the arc list; and (3) the *candidate list pivot rule*, which maintains a candidate list of arcs with positive violation and selects the arc with the maximum violation from the candidate list as the entering arc. We can use these same rules for the generalized network simplex algorithm.

Identifying the Leaving Arc

To describe a procedure for determining the leaving arc, suppose that we select arc (k, l) as the entering arc. The arc (k, l) belongs to the set **L** or the set **U**. Throughout the discussion in this section, we examine the situation in which (k, l) is at its lower bound; we leave the case when (k, l) is at its upper bound as an exercise for the reader (see Exercise 15.22). The approach first determines the rate by which the flow on any arc (i, j) of **F** changes per unit increase of flow on the entering arc (k, l). Let y_{ij} denote this rate. It is easy to verify that the flow on the augmented forest arcs change linearly with the flow on the entering arc (k, l) [i.e., if the entering arc (k, l) carries δ units of flow, the flow change on arc (i, j) is δy_{ij}. Therefore, if we know the y_{ij} values, we can easily compute the maximum value of δ for which all the arc flows remain within their lower and upper bounds. At this value of δ, at least one arc in **F** reaches its lower or upper bound and we select one such arc as the leaving arc.

We now address the problem of determining the y_{ij} values, which represent the flow changes produced on the arcs when we send 1 unit of flow on the entering arc (k, l) (i.e., set $y_{kl} = 1$). We can compute the other y_{ij} values by setting the flow on arc (k, l) equal to 1 and then determining flows on the other arcs so that every node satisfies the mass balance constraint. Setting the flow on arc (k, l) to value 1 creates a deficit (or, demand) of 1 unit at node k and an excess (or, supply) of μ_{kl} units at node l. To determine the effect of sending a unit flow on arc (k, l), we apply the procedure *compute-flows* on the augmented trees containing nodes k and l, starting with zero flow and the following imbalance vector:

$$e(i) = \begin{cases} -1 & \text{for } i = k, \\ \mu_{kl} & \text{for } i = l, \\ 0 & \text{for all } i \neq k \text{ and } l. \end{cases}$$

If the nodes k and l belong to different augmented trees, we need to execute the procedure *compute-flows* twice; otherwise, one execution is sufficient. The arc flows we obtain are the y_{ij} values. Having determined the y_{ij} values, we can easily

compute the maximum additional flow δ on the entering arc (k, l). Let x_{ij} denote the flow corresponding to the current augmented forest structure (**F**, **L**, **U**). The flow bound constraints require that

$$0 \leq x_{ij} + \delta y_{ij} \leq u_{ij} \qquad \text{for all } (i, j) \in \mathbf{F} \cup \{(k, l)\}.$$

If for some arc (i, j), $y_{ij} > 0$, the flow on the arc increases with δ and will eventually reach the arc's upper bound. Similarly, if $y_{ij} < 0$, the flow on the arc decreases with δ and will eventually reach its lower bound. Therefore, if δ_{ij} denotes the maximum possible increase in δ allowed by arc (i, j), then

$$\delta_{ij} = \begin{cases} (u_{ij} - x_{ij})/y_{ij} & \text{if } y_{ij} > 0, \\ x_{ij}/(-y_{ij}) & \text{if } y_{ij} < 0, \\ \infty & \text{if } y_{ij} = 0. \end{cases}$$

The largest value of δ for which $x + \delta y$ is feasible is

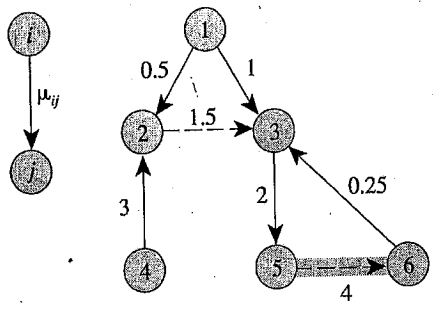
$$\delta = \min[\delta_{ij} : (i, j) \in \mathbf{F} \cup \{(k, l)\}].$$

We next augment δ units of flow on the arc (k, l) and change the flow on the augmented forest arcs to $x_{ij} + \delta y_{ij}$. We refer to any arc (i, j) that defines δ, that is, for which $\delta = \delta_{ij}$, as a *blocking arc*. We select any blocking arc, say (p, q), as the leaving arc. Observe that for the leaving arc (p, q), $y_{pq} \neq 0$. We say that the iteration is *nondegenerate* if $\delta > 0$, and *degenerate* if $\delta = 0$. A degenerate iteration occurs only if **F** is a degenerate augmented forest.

We illustrate the procedure of determining the leaving arc on a numerical example. Figure 15.12(a) shows an augmented tree along with the arc multipliers. Figure 15.12(b) gives the arc capacities and the current arc flows. Assume that (5, 6) is the entering arc. We first determine the values y_{ij} of the flow changes. To determine these values, we set the flow on the entering arc to value 1 and the flow on the extra arc (2, 3) to value θ. Then, as described in the procedure *compute-flows*, we examine the leaf nodes of the tree, one by one, and determine the flows on the unique arcs incident to them. Figure 15.12(c) gives the arc flows as a function of θ. The excess at the root node 1 is $0.5 - 0.5\theta$ and setting this quantity equal to 0 gives $\theta = 1$. Figure 15.12(b) shows the resulting numerical value for each y_{ij}. This figure also specifies the values of each δ_{ij}; using these values, we find that $\delta = 2$. Clearly, (1, 3) is the leaving arc and dropping it gives the augmented tree shown in Figure 15.12(d) with arc (5, 6) as the extra arc.

Updating the Augmented Forest

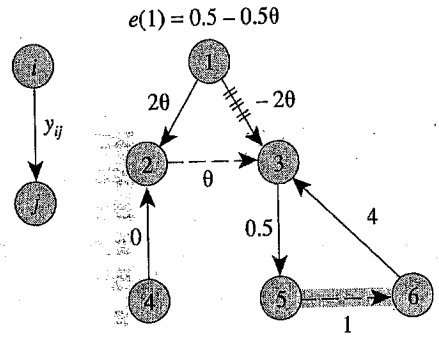
When the generalized network simplex algorithm has determined a leaving arc (p, q) for a given entering arc (k, l), it updates the augmented forest structure. If the leaving arc is the same as the entering arc, which would happen when $\delta = \delta_{kl} = u_{kl}$, the augmented forest does not change. 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 augmented forest changes and we need to update the sets **F**, **L**, and **U**. In this case we need to show that the modified set of arcs in **F** constitutes an augmented forest and is good. To do so, we use the fact that the simplex method moves from one basis to another. Since each basis in the generalized network flow



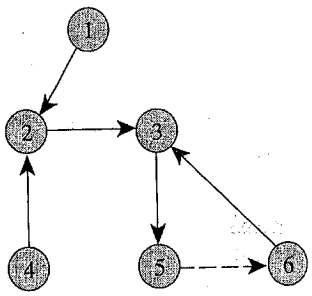
(a)

(i, j)	(1, 2)	(1, 3)	(2, 3)	(3, 5)	(4, 2)	(5, 6)	(6, 3)
μ_{ij}	0.5	1	1.5	2	3	4	0.25
u_{ij}	10	∞	8	∞	5	7	∞
x_{ij}	5	4	3	2	0	2	5
y_{ij}	2	–2	1	0.5	0	1	4
δ_{ij}	2.5	2	5	∞	∞	5	∞

(b)



(c)



(d)

Figure 15.12 Illustrating the selection of a leaving arc.

problem is a good augmented forest, the generalized network simplex algorithm moves from one good augmented forest to another good augmented forest.

Updating Potentials and Tree Indices

After we have updated the flows and obtained a new augmented forest structure, the next step is to update the node potentials. Clearly, we need to update the node potentials for only those nodes that belong to the augmented tree(s) involved in the pivot operation. In fact, for the generalized network flow problem, updating the node potentials appears to be almost as difficult as recomputing them from scratch. We can recompute the node potentials of the augmented tree(s) using the procedure *compute-potentials* described in Figure 15.6. The final step in the pivot operation is to update various tree indices. This step is rather involved and for details we refer the reader to the references given in the reference notes. Alternatively, we could compute the tree indices from scratch, which would also require $O(n)$ time.

Flows in Bicycles

As we have seen in Chapter 11, in the process of moving from one spanning tree solution to another, the network simplex algorithm sends flows along cycles. Since this interpretation helped us to understand the network simplex algorithm, we might

naturally ask the following question: What is the counterpart of a cycle in the generalized network flow problem? Alternatively, we might pose this question as follows. Let y_{ij} be the changes in the flows on the arcs that we defined previously, and let Y denote the set of arcs with strictly positive values of y_{ij}. Then, what is the graph-theoretic structure of Y?

To answer this question, consider several possibilities for the entering arc (k, l) as shown in Figure 15.13. Figure 15.13(a) shows a case when the entering arc has both of its endpoints in different augmented trees and Figure 15.13(b) to (d) show three cases when the entering arc has both of its endpoints in the same augmented

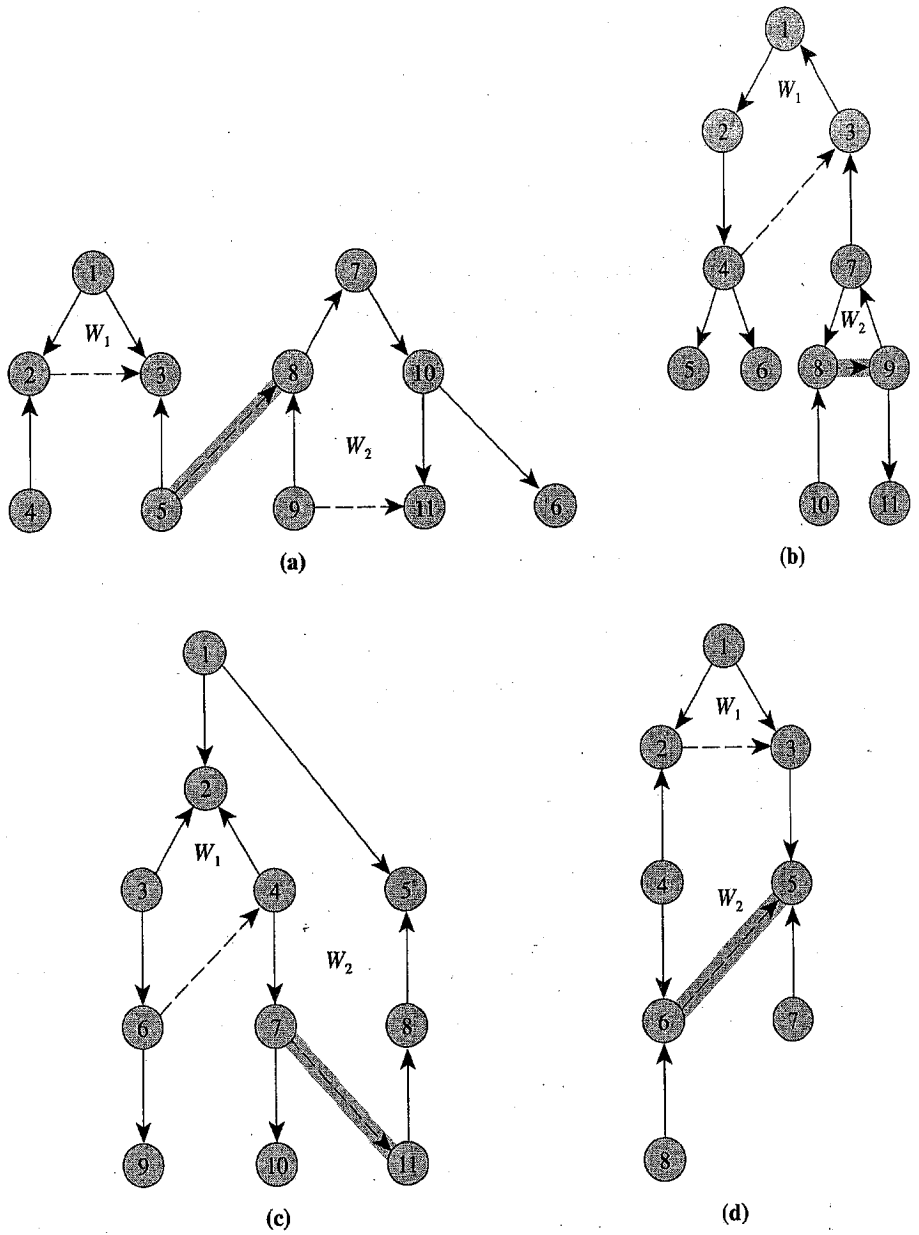


Figure 15.13 Several possibilities for the entering arc.

tree. Consider Figure 15.13(a) first. To increase the flow on the entering arc (5, 8), we first circulate some flow along W_1 starting at some node, say node 3, in a direction defined so that W_1 is a gainy cycle. This flow change creates some excess at node 3. We send this excess to node 5 by the unique tree path, and then to node 8 through the arc (5, 8). Finally, we send a sufficient amount of flow along the cycle W_2 (whose direction is defined so that it is a lossy cycle), so that the excess available at node 8 is consumed. Clearly, y_{ij} will be positive for any arc in the cycles W_1 and W_2 and in the unique path connecting these two cycles. The flow changes in the cases shown in Figure 15.13(b) to (d) have similar interpretations. In each case we have two cycles: one is gainy and the other is lossy. We create an excess by sending flow along one cycle and consume it by circulating flow along the second cycle. As a result, we increase the flow on the entering arc and satisfy the mass balance constraints at all the nodes.

The preceding observations imply that the subgraph defined by the arcs in the set Y is one of the two types shown in Figure 15.14. We refer to the subgraph shown in Figure 15.14(a) as a *type 1 bicycle* and the subgraph shown in Figure 15.14(b) as a *type 2 bicycle*. Figure 15.13 illustrates these bicycles. The cases shown in Figure 15.13(a) and (b) have type 1 bicycles and the cases shown in Figure 15.13(c) and (d) have type 2 bicycles. Therefore, bicycles play the same role in the generalized network simplex algorithm as cycles play in the network simplex algorithm.

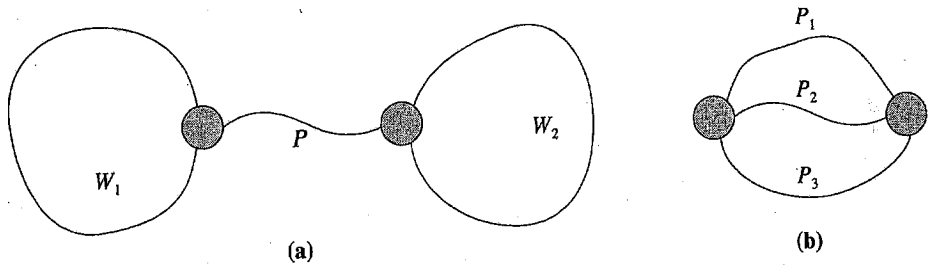


Figure 15.14 Two types of bicycles.

Finally, we note that in the generalized network simplex algorithm, the number of augmented trees in **F** might change from one iteration to the next. For example, consider the situations shown in Figure 15.13(a). If arc (7, 8) is the leaving arc, the pivot creates one fewer augmented tree. On the other hand, if (5, 3) is the leaving arc, the pivot does not change the number of augmented trees. Consider the situation shown in Figure 15.13(b). In this case, if (7, 3) is the leaving arc, the pivot creates an additional augmented tree; for every other leaving arc, the number of augmented trees remain unchanged.

Termination

The generalized network simplex algorithm moves from one feasible augmented forest structure to another until it obtains a structure that satisfies the optimality conditions. In a pivot operation, the objective function value decreases by the

amount $\delta \mid c_{kl}^{\pi} \mid$. If each pivot operation in the algorithm is nondegenerate (i.e., $\delta > 0$), each subsequent augmented forest structure has a smaller cost. Since any network has a finite number of augmented forest structures and each augmented forest structure has a unique associated cost, the generalized network simplex algorithm terminates finitely. Degenerate pivots, however, pose some difficulties: The algorithm might not terminate finitely unless we perform pivots carefully. In the following discussion we describe a perturbation technique that ensures that the generalized network simplex algorithm always terminates finitely.

Complexity

The worst-case complexity of the generalized network simplex algorithm is the product of the number of iterations and the complexity per iteration. Although in the worst case we cannot bound the number of iterations by any polynomial function of n and m, in practice the number of iterations an algorithm performs is generally a low-order polynomial in n and m. And what is the running time per iteration? The algorithm requires $O(m)$ time to identify the entering arc. Each of the other operations, such as computing the y_{ij} values, updating the flows, the potentials, and the tree indices, requires $O(m)$ time. Therefore, in practice, the running time of the generalized network simplex algorithm is a low-order polynomial of n and m. Empirical investigations have found that the generalized network simplex algorithm is only two to three times slower than the network simplex algorithm for the minimum cost flow problem.

Perturbation

To ensure finite termination of the algorithm for problems that are degenerate, we use the well-known perturbation technique of linear programming, which we discussed for the minimum cost flow problem in Exercise 11.26. For the generalized network flow problem, we define ϵ as an n-vector whose elements are $(\alpha, \alpha, \alpha, \ldots, \alpha)$ for a sufficiently small real number α. We then replace the supply/demand vector b by $b + \epsilon$ and apply the generalized network simplex algorithm to the perturbed problem. It is possible to show that for the perturbed problem, the augmented forest structure maintained by the generalized simplex algorithm at every iteration is nondegenerate. Consequently, every pivot is nondegenerate and the algorithm will terminate finitely. Moreover, it is possible to show that an optimal augmented forest structure of the perturbed problem is also an optimal augmented forest structure of the original problem. Thus the perturbation does not affect optimality of the solution. For the sake of brevity, we do not specify the details of these results (see the reference notes for a citation to the literature). We do note, however, that we do not need to actually carry out the perturbation. Just as the perturbation for the network simplex algorithm is equivalent to maintaining strongly feasible spanning tree structures, similarly, perturbation in the generalized network simplex algorithm reduces to maintaining certain special types of augmented forest structures which are called *strongly feasible* augmented forest structures (see the reference notes for citations to the literature concerning this issue as well).

15.7 SUMMARY

The generalized network flow problem is a generalization of the minimum cost flow problem in the sense that arcs do not conserve flow. The generalized network flow problem is significantly more difficult to solve than the minimum cost flow problem and, in only a few instances, can we generalize algorithms for the minimum cost flow algorithm so that they would be suitable for solving the generalized network flow problem. The network simplex algorithm discussed in Chapter 11 is one such instance and the resulting algorithm, called the *generalized network simplex algorithm*, is the fastest available algorithm for solving the generalized network flow problem in practice. This chapter has developed the details of the generalized network simplex algorithm.

The generalized network simplex algorithm is an adaptation of the linear programming simplex method (see Appendix C) for the generalized network flow problem. This adaptation is possible because of the special topological structure of the basis. The basis of the generalized network flow problem is a good augmented forest; this fact permits us to perform the steps of the simplex method without maintaining the simplex tableau. Good augmented forests play the same role in the generalized network simplex algorithm as do spanning trees in the network simplex algorithm. The optimality conditions for a good augmented forest are the same as those for the minimum cost flow problem, but with a slightly different definition of the reduced cost c_{ij}^{π} of an arc (i, j); in this context, it is $c_{ij}^{\pi} = c_{ij} - \pi(i) + \mu_{ij}\pi(j)$.

The generalized network simplex algorithm performs two fundamental operations at every iteration: determining the node potentials and arc flows associated with a good augmented forest structure. We showed how to implement both operations very efficiently, in $O(m)$ time, using methods that generalize their counterparts in the network simplex algorithm.

The generalized network simplex algorithm maintains a feasible (good) augmented forest structure at every iteration and successively transforms it into an improved augmented forest structure until it becomes optimal. To implement the generalized network simplex algorithm, we can use the tree indices that we described in Chapter 11 in our discussion of the implementation of the network simplex algorithm. Using these indices permits us to select the leaving variable and to update the flows and potentials in $O(n)$ time. The time to select an entering arc is $O(m)$. The generalized network simplex algorithm terminates finitely, even though the number of iterations it performs cannot be bounded by a polynomial or pseudopolynomial function of the input size parameters. In practice, however, because the algorithm rarely performs more than $5m$ iterations for moderately sized problems and because the average time per pivot is much less than n, its computational time grows slower than $O(nm)$, even though its worst-case running time is exponential. Empirical tests have found the generalized network simplex algorithm to be about two to three times slower than the network simplex algorithm.

REFERENCE NOTES

The generalized network simplex algorithm, presented in this chapter, is an adaptation of the linear programming simplex method and is due to Dantzig [1962]. Kennington and Helgason [1980] and Jensen and Barnes [1980] have given other textbook

treatments of the generalized network simplex algorithm. Our presentation of the generalized network simplex algorithm differs from these presentations; it is more combinatorial than algebraic. The approach we have adopted appears for the first time in this book. Elam, Glover, and Klingman [1979] describe the generalized network simplex algorithm, which maintains a strongly feasible basis at every step. Orlin [1985] discusses pertubation techniques for the generalized network simplex algorithm. Elam, Glover, and Klingman [1979] and Brown and McBride [1984] have presented implementation details of the generalized network simplex algorithm and have examined the computational performances of the resulting algorithms. These investigations have found that the generalized network simplex algorithm is about two to three times slower than the network simplex algorithm for the minimum cost flow problem, but substantially faster than a general-purpose linear programming code.

Researchers have also studied nonsimplex approaches for the generalized network flow problem. These approaches include (1) a primal–dual algorithm developed by Jewell [1962], (2) a dual algorithm proposed by Jensen and Bhaumik [1977], and (3) a relaxation algorithm developed by Bertsekas and Tseng [1988b]. Balachandran and Thompson [1975] describe procedures for performing sensitivity and parametric analyses for the generalized network flow problem.

Researchers have also actively studied the generalized maximum flow problem, which is a special case of the generalized network flow problem. These studies have produced simpler algorithms for this problem. The survey paper by Truemper [1977] described these approaches and showed that the generalized maximum flow problem is, in several ways, related to the (ordinary) minimum cost flow problem. None of these algorithms are pseudopolynomial-time algorithms, partly because the optimal arc flows and node potentials might be fractional. Goldberg, Plotkin, and Tardos [1991] presented the first polynomial-time combinatorial algorithm for the generalized maximum flow problem.

In Section 15.2 we described several applications of the generalized network flow problem. Our discussion of these applications has been adapted from the following papers:

1. Conversion of physical entities (Golden, Liberatore, and Lieberman [1979], Glover, Glover, and Shields [1988], and Farina and Glover [1983])
2. Machine loading (Dantzig [1962])
3. Managing warehousing goods and funds flow (Cahn [1948])
4. Land management (Glover, Glover, and Martinson [1984])

In Section 1.3 we described another application of generalized networks arising in energy modeling. Additional applications of the generalized network flow problem arise in (1) resort development (Glover and Rogozinski [1982]), (2) airline seat allocation problems (Dror, Trudeau, and Ladany [1988]), (3) personnel planning (Gorham [1963]), (4) a consensus ranking model (Barzilai, Cook, and Kress [1986]), and (5) cash flow management in an insurance company (Crum and Nye [1981]). The survey papers of Glover, Hultz, Klingman, and Stutz [1978] and Glover, Klingman, and Phillips [1990] contain additional references concerning applications of generalized network flow problems.

EXERCISES

15.1. Generalized caterer problem. In Exercise 11.2 we studied a particular version of the caterer problem; here we consider a generalization of this problem. Suppose that two types of laundry service are available, slow and fast. Suppose, in addition, that each type of service incurs a loss: the slow service loses 5 percent of the napkins it is given to clean and a fast service loses 10 percent of the napkins it is given. The objective, as earlier, is to provide the desired number of napkins on each week day at the lowest possible total cost. Show how to model this extension of the caterer problem as a generalized network flow problem.

15.2. Production scheduling problem. A steel fabricator has several manufacturing plants. Plant i has a manufacturing capacity of S_i tons per month. The company produces n distinct products and for a given month the total customer demand for product j is D_j tons. The plants differ in their fabricating facilities and production efficiencies. A ton of capacity at plant i can produce a_{ij} tons of product j. The fabricator incurs a cost of c_{ij} dollars for each ton of product j produced at plant i and would like to allocate its customer demands to its plants at the least possible cost. Formulate this problem as a generalized network flow problem.

15.3. Formulate the aircraft assignment problem described in Application 15.2 as a generalized network flow problem.

15.4. Optimal currency conversion. Each day an exchange control bureau permits people to exchange a limited amount of money from one currency to another. On a particular day, suppose that the table shown in Figure 15.15 specifies these limits as well as the exchange rates of various currencies that the exchange control department handles. Suppose that you have \$1000 and you want to determine the maximum number of francs you can obtain on that day through a sequence of currency transactions. Formulate this problem as a generalized maximum flow problem.

Currency given, x	Currency received, y	Exchange rate, r ($y = rx$)	Limit (on x)
Dollars	Pounds	0.56	1,000
Dollars	Lira	1,241	500
Pounds	Lira	2,200	160
Lira	Pounds	0.00045	200,000
Guilders	Pounds	3.37	400
Lira	Yen	0.11	950,000
Yen	Guilders	0.014	15,000
Guilders	Yen	70.5	500
Guilders	Francs	3.0	1,600
Yen	Francs	0.042	80,000

Figure 15.15 Currency exchange rates on a particular day.

15.5. In Exercise 11.3 we studied a project assignment problem in which each project had exactly one supervisor. Suppose that we permit each project to have a group of supervisors and we know this group for each project. Show that it is difficult to model

the modified problem as a minimum cost flow problem but that we can model it as an integer generalized network flow problem.

15.6. In this exercise we study a few generalizations of the warehouse funds and goods flows problem that we discussed in Application 15.3. Give network flow formulations for situations with $K = 4$ time periods when we impose the following additional problem features.

(a) At the beginning of each period, the entrepreneur can lend cash for 2 or 3 years, accruing an interest of 20 percent and 35 percent.

(b) At the beginning of each period, the entrepreneur can borrow cash for 2 years. She pays an interest in the amount of 25 percent for the 2-year period.

(c) The entrepreneur must satisfy a demand of $d(k)$ units for the product in each time period k.

15.7. In the warehouse funds and goods flows model that we discussed in Application 15.3, suppose that the warehouse can store two products: In every period the entrepreneur can convert portions of each product into cash or into each other. The conversion ratios for each period are known in advance. Can you formulate this model as a generalized network flow problem? If so, give the formulation; if no, outline the difficulties encountered.

15.8. Give two real-life situations, not described in this chapter, of networks that have flow gains and/or losses on their arcs.

15.9. Explain how you would model each of the following extensions of the generalized network flow problems: (1) the flow on arc (i, j) has a nonnegative lower flow bound l_{ij}; (2) the multiplier of an arc (i, j) is zero; (3) a supply node i is permitted to keep some of its supply $b(i)$; and (4) at most $u(i)$ units can enter node i. Consider each of these generalizations separately.

15.10. **(a)** Consider a linear programming problem in which each column has exactly two nonzero entries: one positive and the other negative. Transform this linear programming problem into a generalized network flow problem.

(b) Consider a linear programming problem satisfying two properties: (1) each column has at most two nonzero entries, and (2) if any column has exactly two nonzero entries, one of these is positive and the other is negative. Transform this linear programming problem into a generalized network flow problem.

15.11. Prove Properties 15.3 and 15.4.

15.12. Show that the generalized network flow problem satisfies $c^{\pi}x = cx - \pi b$ for any π. Conclude that any solution that minimizes cx also minimizes $c^{\pi}x$. (*Hint*: The proof is similar to that of property 2.4.)

15.13. In the generalized network simplex algorithm, we obtain an initial augmented forest structure by introducing an artificial arc (i, i), with a sufficiently large cost M, for every node $i \in N$. Specify a finite value of M, as a function of the problem data, that would ensure that these artificial arcs carry zero flow if the generalized network flow problem has a feasible solution. (*Hint*: Determine an upper bound α on the objective function value of any feasible flow without using artificial arcs. Next, determine a lower bound β on the flow on any artificial arc (i, i) if that arc carries a positive flow in any augmented forest. Select M so that $M\beta > \alpha$.)

15.14. Professor May B. Wright announced a novel algorithm for solving the generalized network flow problem when all the arcs are lossy. Can you transform a generalized network flow problem with arbitrary positive arc multipliers to a problem in which all arcs are lossy? Assume that all arcs have finite capacities.

15.15. Suppose that we associate a positive real number $\alpha(i)$ with each node $i \in N$ in a graph G. With respect to the vector α, we define the *reduced multiplier* of an arc (i, j) as $\mu_{ij}^{\alpha} = \alpha(i)\mu_{ij}/\alpha(j)$.

(a) Show that $\mu^{\alpha}(W) = \mu(W)$ for any cycle W.

(b) Let T be a spanning tree of G. Define a vector α so that $\mu_{ij}^{\alpha} = 1$ for every arc $(i, j) \in T$. Let $W(k, l)$ be the fundamental cycle defined by a nontree arc (k, l).

Show that if $W(k, l)$ is a breakeven cycle, $\mu_{kl}^{\alpha} = 1$. Conclude that if every fundamental cycle of G with respect to T is a breakeven cycle, then $\mu_{ij}^{\alpha} = 1$ for all $(i, j) \in A$.

(c) Prove that all cycles in G are breakeven if and only if all fundamental cycles with respect to any spanning tree T are breakeven. Use this result to show that we can determine in $O(m)$ time whether all cycles in G are breakeven.

15.16. Consider the linear programming formulation of the generalized network flow problem given in (15.1). Suppose that we associate a positive real number $\alpha(i)$ with each node $i \in N$ and define the new set of variables y as $y_{ij} = \alpha(i)x_{ij}$. Suppose that we then make the substitution $x_{ij} = y_{ij}/\alpha(i)$ for every (i, j) in (15.1). Show that the resulting formulation is also a generalized network flow problem, but with different supplies/demands, arc costs, capacities, and multipliers. We refer to the resulting problem as the *α-transformed problem.*

(a) What are the supplies/demands, arc costs, capacities, and multipliers of the α-transformed problem in terms of α and the original problem data?

(b) Show that if all the cycles in G are breakeven, then for some choice of the vector α, the α-transformed problem becomes a minimum cost flow problem. (*Hint*: Use the result of Exercise 15.15.)

15.17. Using the procedure *compute-potentials*, determine node potentials for the augmented trees shown in Figure 15.16.

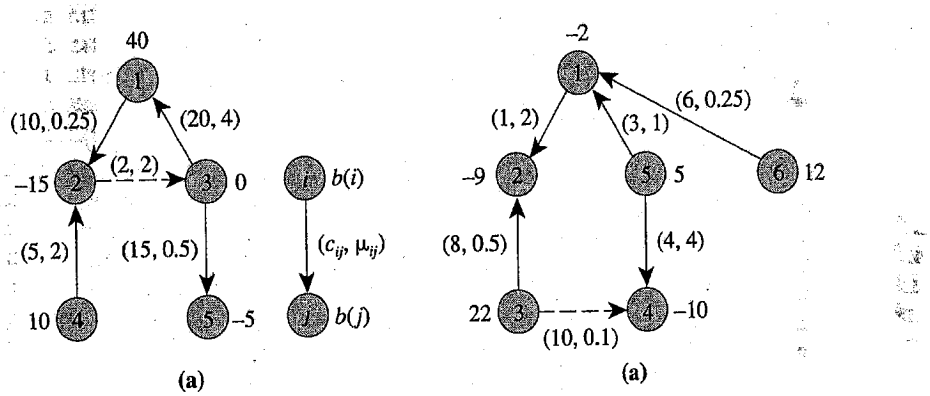


Figure 15.16 Two augmented trees.

15.18. Using the procedure *compute-flows*, determine arc flows for the augmented trees shown in Figure 15.16. Assume that all arcs are uncapacitated. Are the flows feasible?

15.19. Apply two iterations of the generalized network simplex algorithm to the generalized network flow problem shown in Figure 15.17. Assume that all arcs are uncapacitated.

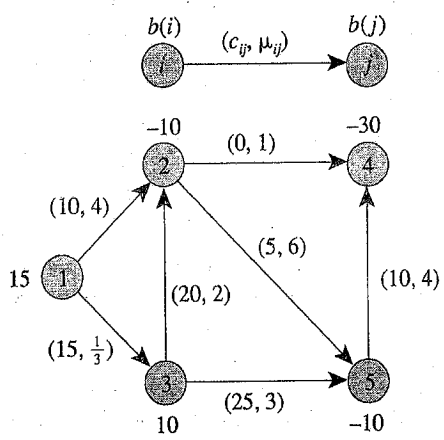


Figure 15.17 Generalized network example for Exercise 15.19.

15.20. Let $T \cup \{(\alpha, \beta)\}$ be an augmented tree of G. Suppose that we define the multiplier of a path P as $\mu(P) = (\sum_{(i,j)\in \overline{P}} \mu_{ij})/(\sum_{(i,j)\in \underline{P}} \mu_{ij})$; in this expression, P and $\underline{P}$ denote the sets of forward and backward arcs in the path P.

(a) Let P_i denote the unique path from any node i to the root node h in T [i.e., the augmenting tree minus the extra arc (α, β)]. Show that at the termination of the procedure *compute-potentials* described in Figure 15.6, each $g(i)$ equals $\mu(P_i)$.

(b) Show that the expression $(g(\alpha) - \mu_{\alpha\beta} g(\beta))$ is nonzero if and only if the augmented tree $T \cup \{(\alpha, \beta)\}$ is good (i.e., contains no breakeven cycle).

15.21. Show that if an augmented forest is not good, node potentials determined by the procedure *compute-potentials* might not be unique. Next argue that if an augmented forest is good, node potentials determined by the procedure *compute-potentials* are unique. Conclude that the procedure *compute-potentials* gives a unique set of node potentials if and only if the augmented forest is good.

15.22. In the description of the generalized network simplex algorithm, we have assumed that the entering arc (k, l) belongs to $\mathbf{L}$. If arc $(k, l) \in \mathbf{U}$, what steps of the algorithm would we need to modify, and how?

15.23. When we add the entering arc (k, l) to an augmented forest, we create a bicycle. Write separate pseudocodes to accomplish the following tasks: (1) determine whether the bicycle is a type 1 bicycle or a type 2 bicycle; (2) if the bicycle is a type 1 bicycle, determine the cycles W_1, W_2 and the path segment P; (3) if the bicycle is a type 2 bicycle, determine the three path segments P_1, P_2, and P_3; (4) if the bicycle is a type 2 bicycle and contains a breakeven cycle, determine the breakeven cycle. Your pseudocodes should use only predecessor and depth indices and run in $O(n)$ time.

15.24. For a given feasible flow x of the generalized network flow problem, we define a *residual network* $G(x)$ as follows. We replace each arc (i, j) by two arcs (i, j) and (j, i). Arc (i, j) has cost c_{ij}, a residual capacity $r_{ij} = u_{ij} - x_{ij}$, and a multiplier μ_{ij}; arc (j, i) has cost $-c_{ij}/\mu_{ij}$, a residual capacity x_{ij}/μ_{ij}, and a multiplier $1/\mu_{ij}$. The residual network consists of only those arcs with a positive residual capacity.

(a) Define the *length* of each arc (i, j) in $G(x)$ as $-\log(\mu_{ij})$. Show that a directed cycle W is a gainy cycle if and only if its length is a negative cycle. Using this result, describe a polynomial-time algorithm for identifying whether $G(x)$ contains a directed cycle that is gainy.

(b) Describe a polynomial-time algorithm for identifying whether $G(x)$ contains a directed cycle that is lossy.

15.25. Generalized maximum flow problem. In the generalized maximum flow problem, we wish to obtain a feasible flow that maximizes one of two objectives: (1) the flow into a sink node t, or (2) the flow out of the source node s. Formulate the generalized maximum flow problem as a generalized minimum cost flow problem. (*Hint*: Transform the problem into one in which all the nodes have zero supplies/demands.)

15.26. Generalized maximum flow algorithm. In this exercise we discuss an algorithm for maximizing the flow into the sink node t. Suppose that we define the residual network $G(x)$ with respect to a flow x as in Exercise 15.24. In the residual network, we define a *generalized augmenting path* as either (1) a directed path from node s to node t, or (2) a directed cycle W that is gainy, plus a directed path from some node in W to node t.

(a) Show that if $G(x)$ contains a generalized augmenting path, we can increase the flow into the sink. Use this result to describe an algorithm for the generalized maximum flow problem.

(b) Describe a polynomial-time algorithm for identifying a generalized augmenting path. (*Hint*: To identify the second type of augmenting path, let G' be the subgraph of G consisting of those nodes that have directed paths to node t. Then look for a gainy cycle in G'.)

15.27. Generalized flow decomposition property. We refer to a generalized network flow problem as a *generalized circulation problem* if $b(i) = 0$ for every node $i \in N$ (i.e., for each node the inflow equals its outflow). In a generalized circulation problem, we refer

to any flow x_{ij} satisfying the mass balance and flow bound constraints as a generalized circulation. Two types of generalized circulation are of special interest: *cycle flow* and *bicycle flow*. A cycle flow is a generalized circulation for which $x_{ij} > 0$ only along arcs of a breakeven cycle; a bicycle circulation is a generalized flow for which $x_{ij} > 0$ only along arcs of a bicycle (of either type 1 or type 2). We refer to a cycle or bicycle flow x as *negative* if $\sum_{(i,j)\in A} c_{ij}x_{ij}$ is negative. Show that it is possible to decompose any generalized circulation into flows along at most m cycle or bicycle flows. (*Hint*: Generalize the method we described in the proof of Theorem 3.5 for identifying a breakeven cycle or a bicycle.)

15.28. Generalized flow optimality conditions. Show that a generalized circulation x^* is an optimal solution of the generalized circulation problem if and only if the network contains no negative cycle or bicycle flow with respect to x^*.

15.29. Show that by adding a loop with a multiplier of $\frac{1}{2}$ to every node of a network, we can formulate any generalized flow problem with one or more inequalities for supplies and demands (i.e., the mass balance constraints are stated as "$\le b(i)$" for a supply node i, and/or "$\ge b(j)$" for a demand node j) into an equivalent problem with all equality constraints (i.e., "$= b(k)$" for all nodes k).

16

LAGRANGIAN RELAXATION AND NETWORK OPTIMIZATION

I never missed the opportunity to remove obstacles in the way of unity.
—Mohandas Gandhi

Chapter Outline

16.1 INTRODUCTION

As we have noted throughout our discussion in this book, the basic network flow models that we have been studying—shortest paths, maximum flows, minimum cost flows, minimum spanning trees, matchings, and generalized and convex flows—arise in numerous applications. These core network models are also building blocks for many other models and applications, in the sense that many models met in practice have embedded network structure: that is, the broader models are network problems with additional variables and/or constraints.

In this chapter we consider ways to solve these models using a solution strategy known as *decomposition* which permits us to draw upon the many algorithms that we have developed in previous chapters to exploit the underlying network structure. In a sense this chapter serves a dual purpose. First, it permits us to introduce a broader set of network optimization models than we have been considering in our earlier discussion. As such, this chapter provides a glimpse of how network flow models arise in a wide range of applied problem settings that cannot be modeled as pure network flow problems. Second, the chapter introduces a solution method, known as *Lagrangian relaxation*, that has become one of the very few solution methods in optimization that cuts across the domains of linear and integer programming, combinatorial optimization, and nonlinear programming.

Perhaps the best way to understand the basic idea of Lagrangian relaxation is via an example.