6

MAXIMUM FLOWS: BASIC IDEAS

*You get the maxx for the minimum at T. J. Maxx.**
—Advertisement for a clothing store.

Chapter Outline

6.1 INTRODUCTION

The maximum flow problem and the shortest path problem are complementary. They are similar because they are both pervasive in practice and because they both arise as subproblems in algorithms for the minimum cost flow problem. The two problems differ, however, because they capture different aspects of the minimum cost flow problem: Shortest path problems model arc costs but not arc capacities; maximum flow problems model capacities but not costs. Taken together, the shortest path problem and the maximum flow problem combine all the basic ingredients of network flows. As such, they have become the nuclei of network optimization. Our study of the shortest path problem in the preceding two chapters has introduced us to some of the basic building blocks of network optimization, such as distance labels, optimality conditions, and some core strategies for designing iterative solution methods and for improving the performance of these methods. Our discussion of maximum flows, which we begin in this chapter, builds on these ideas and introduces several other key ideas that reoccur often in the study of network flows.

The *maximum flow problem* is very easy to state: In a capacitated network, we wish to send as much flow as possible between two special nodes, a source node s and a sink node t, without exceeding the capacity of any arc. In this and the following two chapters, we discuss a number of algorithms for solving the maximum flow problem. These algorithms are of two types:

* © The TJX Operating Companies, Inc. 1985. Get the Maxx for the Minimum® and T. J. Maxx are registered trademarks of the TJX Operating Companies, Inc.

1. Augmenting path algorithms that maintain mass balance constraints at every node of the network other than the source and sink nodes. These algorithms incrementally augment flow along paths from the source node to the sink node.
2. Preflow-push algorithms that flood the network so that some nodes have excesses (or buildup of flow). These algorithms incrementally relieve flow from nodes with excesses by sending flow from the node forward toward the sink node or backward toward the source node.

We discuss the simplest version of the first type of algorithm in this chapter and more elaborate algorithms of both types in Chapter 7. To help us to understand the importance of the maximum flow problem, we begin by describing several applications. This discussion shows how maximum flow problems arise in settings as diverse as manufacturing, communication systems, distribution planning, matrix rounding, and scheduling.

We begin our algorithmic discussion by considering a *generic augmenting path algorithm* for solving the maximum flow problem and describing an important special implementation of the generic approach, known as the *labeling algorithm*. The labeling algorithm is a pseudopolynomial-time algorithm. In Chapter 7 we develop improved versions of this generic approach with better theoretical behavior. The correctness of these algorithms rests on the renowned *max-flow min-cut theorem* of network flows (recall from Section 2.2 that a cut is a set of arcs whose deletion disconnects the network into two parts). This central theorem in the study of network flows (indeed, perhaps the most significant theorem in this problem domain) not only provides us with an instrument for analyzing algorithms, but also permits us to model a variety of applications in machine and vehicle scheduling, communication systems planning, and several other settings, as maximum flow problems, even though on the surface these problems do not appear to have a network flow structure. In Section 6.6 we describe several such applications.

The max-flow min-cut theorem establishes an important correspondence between flows and cuts in networks. Indeed, as we will see, by solving a maximum flow problem, we also solve a complementary *minimum cut problem*: From among all cuts in the network that separate the source and sink nodes, find the cut with the minimum capacity. The relationship between maximum flows and minimum cuts is important for several reasons. First, it embodies a fundamental duality result that arises in many problem settings in discrete mathematics and that underlies linear programming as well as mathematical optimization in general. In fact, the max-flow min-cut theorem, which shows the equivalence between the maximum flow and minimum cut problems, is a special case of the well-known strong duality theorem of linear programming. The fact that maximum flow problems and minimum cut problems are equivalent has practical implications as well. It means that the theory and algorithms that we develop for the maximum flow problem are also applicable to many practical problems that are naturally cast as minimum cut problems. Our discussion of combinatorial applications in the text and exercises of this chapter and our discussion of applications in Chapter 19 features several applications of this nature.

Notation and Assumptions

We consider a capacitated network $G = (N, A)$ with a *nonnegative* capacity u_{ij} associated with each arc $(i, j) \in A$. Let $U = \max\{u_{ij}: (i, j) \in A\}$. As before, the arc adjacency list $A(i) = \{(i, k): (i, k) \in A\}$ contains all the arcs emanating from node i. To define the maximum flow problem, we distinguish two special nodes in the network G: a *source node s* and a *sink node t*. We wish to find the maximum flow from the source node s to the sink node t that satisfies the arc capacities and mass balance constraints at all nodes. We can state the problem formally as follows.

$$\text{Maximize } v \tag{6.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} v & \text{for } i = s, \\ 0 & \text{for all } i \in N - \{s \text{ and } t\} \\ -v & \text{for } i = t \end{cases} \tag{6.1b}$$

$$0 \le x_{ij} \le u_{ij} \qquad \text{for each } (i, j) \in A. \tag{6.1c}$$

We refer to a vector $x = \{x_{ij}\}$ satisfying (6.1b) and (6.1c) as a *flow* and the corresponding value of the scalar variable v as the *value* of the flow. We consider the maximum flow problem subject to the following assumptions.

Assumption 6.1. *The network is directed.*

As explained in Section 2.4, we can always fulfill this assumption by transforming any undirected network into a directed network.

Assumption 6.2. *All capacities are nonnegative integers.*

Although it is possible to relax the integrality assumption on arc capacities for some algorithms, this assumption is necessary for others. Algorithms whose complexity bounds involve U assume integrality of the data. In reality, the integrality assumption is not a restrictive assumption because all modern computers store capacities as rational numbers and we can always transform rational numbers to integer numbers by multiplying them by a suitably large number.

Assumption 6.3. *The network does not contain a directed path from node s to node t composed only of infinite capacity arcs.*

Whenever every arc on a directed path P from s to t has infinite capacity, we can send an infinite amount of flow along this path, and therefore the maximum flow value is unbounded. Notice that we can detect the presence of an infinite capacity path using the search algorithm described in Section 3.4.

Assumption 6.4. *Whenever an arc (i, j) belongs to A, arc (j, i) also belongs to A.*

This assumption is nonrestrictive because we allow arcs with zero capacity.

Assumption 6.5. The network does not contain parallel arcs (i.e., two or more arcs with the same tail and head nodes).

This assumption is essentially a notational convenience. In Exercise 6.24 we ask the reader to show that this assumption imposes no loss of generality.

Before considering the theory underlying the maximum flow problem and algorithms for solving it, and to provide some background and motivation for studying the problem, we first describe some applications.

6.2 APPLICATIONS

The maximum flow problem, and the minimum cut problem, arise in a wide variety of situations and in several forms. For example, sometimes the maximum flow problem occurs as a subproblem in the solution of more difficult network problems, such as the minimum cost flow problem or the generalized flow problem. As we will see in Section 6.6, the maximum flow problem also arises in a number of combinatorial applications that on the surface might not appear to be maximum flow problems at all. The problem also arises directly in problems as far reaching as machine scheduling, the assignment of computer modules to computer processors, the rounding of census data to retain the confidentiality of individual households, and tanker scheduling. In this section we describe a few such applications; in Chapter 19 we discuss several other applications.

Application 6.1 Feasible Flow Problem

The feasible flow problem requires that we identify a flow x in a network $G = (N, A)$ satisfying the following constraints:

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i) \qquad \text{for } i \in N, \tag{6.2a}$$

$$0 \le x_{ij} \le u_{ij} \qquad \text{for all } (i, j) \in A. \tag{6.2b}$$

As before, we assume that $\sum_{i\in N} b(i) = 0$. The following distribution scenario illustrates how the feasible flow problem arises in practice. Suppose that merchandise is available at some seaports and is desired by other ports. We know the stock of merchandise available at the ports, the amount required at the other ports, and the maximum quantity of merchandise that can be shipped on a particular sea route. We wish to know whether we can satisfy all of the demands by using the available supplies.

We can solve the feasible flow problem by solving a maximum flow problem defined on an augmented network as follows. We introduce two new nodes, a source node s and a sink node t. For each node i with $b(i) > 0$, we add an arc (s, i) with capacity $b(i)$, and for each node i with $b(i) < 0$, we add an arc (i, t) with capacity $-b(i)$. We refer to the new network as the *transformed network*. Then we solve a maximum flow problem from node s to node t in the transformed network. If the maximum flow saturates all the source and sink arcs, problem (6.2) has a feasible solution; otherwise, it is infeasible. (In Section 6.7 we give necessary and sufficient conditions for a feasible flow problem to have a feasible solution.)

It is easy to verify why this algorithm works. If x is a flow satisfying (6.2a) and (6.2b), the same flow with $x_{si} = b(i)$ for each source arc (s, i) and $x_{it} = -b(i)$ for each sink arc (i, t) is a maximum flow in the transformed network (since it saturates all the source and the sink arcs). Similarly, if x is a maximum flow in the transformed network that saturates all the source and the sink arcs, this flow in the original network satisfies (6.2a) and (6.2b). Therefore, the original network contains a feasible flow if and only if the transformed network contains a flow that saturates all the source and sink arcs. This observation shows how the maximum flow problem arises whenever we need to find a feasible solution in a network.

Application 6.2 Problem of Representatives

A town has r residents $R_1, R_2, \ldots, R_r$; q clubs $C_1, C_2, \ldots, C_q$; and p political parties $P_1, P_2, \ldots, P_p$. Each resident is a member of at least one club and can belong to exactly one political party. Each club must nominate one of its members to represent it on the town's governing council so that the number of council members belonging to the political party P_k is at most u_k. Is it possible to find a council that satisfies this "balancing" property?

We illustrate this formulation with an example. We consider a problem with $r = 7$, $q = 4$, $p = 3$, and formulate it as a maximum flow problem in Figure 6.1. The nodes $R_1, R_2, \ldots, R_7$ represent the residents, the nodes $C_1, C_2, \ldots, C_4$ represent the clubs, and the nodes $P_1, P_2, \ldots, P_3$ represent the political parties.

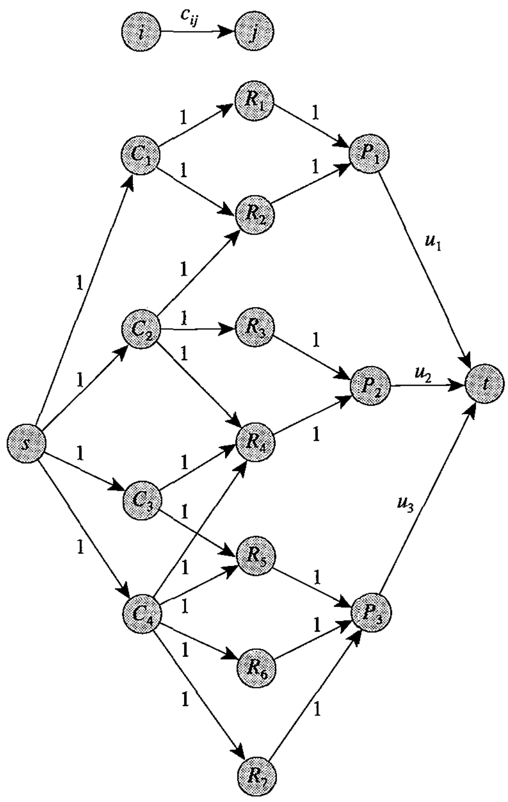


Figure 6.1 System of distinct representatives.

The network also contains a source node s and a sink node t. It contains an arc (s, C_i) for each node C_i denoting a club, an arc (C_i, R_j) whenever the resident R_j is a member of the club C_i, and an arc (R_j, P_k) if the resident R_j belongs to the political party P_k. Finally, we add an arc (P_k, t) for each $k = 1, \ldots, 3$ of capacity u_k; all other arcs have unit capacity.

We next find a maximum flow in this network. If the maximum flow value equals q, the town has a balanced council; otherwise, it does not. The proof of this assertion is easy to establish by showing that (1) any flow of value q in the network corresponds to a balanced council, and that (2) any balanced council implies a flow of value q in the network.

This type of model has applications in several resource assignment settings. For example, suppose that the residents are skilled craftsmen, the club C_i is the set of craftsmen with a particular skill, and the political party P_k corresponds to a particular seniority class. In this instance, a balanced town council corresponds to an assignment of craftsmen to a union governing board so that every skill class has representation on the board and no seniority class has a dominant representation.

Application 6.3 Matrix Rounding Problem

This application is concerned with consistent rounding of the elements, row sums, and column sums of a matrix. We are given a $p \times q$ matrix of *real* numbers $D = \{d_{ij}\}$, with row sums α_i and column sums β_j. We can round any real number a to the next smaller integer $\lfloor a \rfloor$ or to the next larger integer $\lceil a \rceil$, and the decision to round up or down is entirely up to us. The matrix rounding problem requires that we round the matrix elements, and the row and column sums of the matrix so that the sum of the rounded elements in each row equals the rounded row sum and the sum of the rounded elements in each column equals the rounded column sum. We refer to such a rounding as a *consistent rounding*.

We shall show how we can discover such a rounding scheme, if it exists, by solving a feasible flow problem for a network with nonnegative lower bounds on arc flows. (As shown in Section 6.7, we can solve this problem by solving two maximum flow problems with zero lower bounds on arc flows.) We illustrate our method using the matrix rounding problem shown in Figure 6.2. Figure 6.3 shows the maximum flow network for this problem. This network contains a node i corresponding to each row i and a node j' corresponding to each column j. Observe that this network

				Row sum
	3.1	6.8	7.3	17.2
	9.6	2.4	0.7	12.7
	3.6	1.2	6.5	11.3
Column sum	16.3	10.4	14.5	

Figure 6.2 Matrix rounding problem.

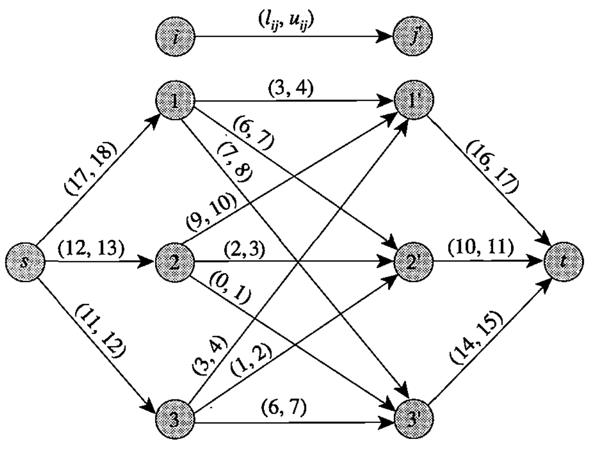


Figure 6.3 Network for the matrix rounding problem.

contains an arc (i, j') for each matrix element d_{ij}, an arc (s, i) for each row sum, and an arc (j', t) for each column sum. The lower and the upper bounds of each arc (i, j') are $\lfloor d_{ij} \rfloor$ and $\lceil d_{ij} \rceil$, respectively. It is easy to establish a one-to-one correspondence between the consistent roundings of the matrix and feasible flows in the corresponding network. Consequently, we can find a consistent rounding by solving a maximum flow problem on the corresponding network.

This matrix rounding problem arises in several application contexts. For example, the U.S. Census Bureau uses census information to construct millions of tables for a wide variety of purposes. By law, the bureau has an obligation to protect the source of its information and not disclose statistics that could be attributed to any particular person. We might disguise the information in a table as follows. We round off each entry in the table, including the row and column sums, either up or down to a multiple of a constant k (for some suitable value of k), so that the entries in the table continue to add to the (rounded) row and column sums, and the overall sum of the entries in the new table adds to a rounded version of the overall sums in the original table. This Census Bureau problem is the same as the matrix rounding problem discussed earlier except that we need to round each element to a multiple of $k \geq 1$ instead of rounding it to a multiple of 1. We solve this problem by defining the associated network as before, but now defining the lower and upper bounds for any arc with an associated real number α as the greatest multiple of k less than or equal to α and the smallest multiple of k greater than or equal to α.

Application 6.4 Scheduling on Uniform Parallel Machines

In this application we consider the problem of scheduling of a set J of jobs on M uniform parallel machines. Each job $j \in J$ has a processing requirement p_j (denoting the number of machine days required to complete the job), a release date r_j (representing the beginning of the day when job j becomes available for processing), and a due date $d_j \geq r_j + p_j$ (representing the beginning of the day by which the job must be completed). We assume that a machine can work on only one job at a time and that each job can be processed by at most one machine at a time. However, we allow *preemptions* (i.e., we can interrupt a job and process it on different machines on different days). The scheduling problem is to determine a feasible schedule that completes all jobs before their due dates or to show that no such schedule exists.

Scheduling problems like this arise in batch processing systems involving batches with a large number of units. The feasible scheduling problem, described in the preceding paragraph, is a fundamental problem in this situation and can be used as a subroutine for more general scheduling problems, such as the maximum lateness problem, the (weighted) minimum completion time problem, and the (weighted) maximum utilization problem.

Let us formulate the feasible scheduling problem as a maximum flow problem. We illustrate the formulation using the scheduling problem described in Figure 6.4 with M = 3 machines. First, we rank all the release and due dates, r_j and d_j for all j, in ascending order and determine $P \leq 2 \mid J \mid - 1$ mutually disjoint intervals of dates between consecutive milestones. Let $T_{k,l}$ denote the interval that starts at the beginning of date k and ends at the beginning of date $l + 1$. For our example, this order of release and due dates is 1, 3, 4, 5, 7, 9. We have five intervals, represented by $T_{1,2}$, $T_{3,3}$, $T_{4,4}$, $T_{5,6}$, and $T_{7,8}$. Notice that within each interval $T_{k,l}$, the set of available jobs (i.e., those released but not yet due) does not change: we can process all jobs j with $r_j \leq k$ and $d_j \geq l + 1$ in the interval.

Job (j)	1	2	3	4
Processing time (p_j)	1.5	1.25	2.1	3.6
Release time (r_j)	3	1	3	5
Due date (d_j)	5	4	7	9

Figure 6.4 Scheduling problem.

We formulate the scheduling problem as a maximum flow problem on a bipartite network G as follows. We introduce a source node s, a sink node t, a node corresponding to each job j, and a node corresponding to each interval $T_{k,l}$, as shown in Figure 6.5. We connect the source node to every job node j with an arc with capacity p_j, indicating that we need to assign p_j days of machine time to job j. We connect each interval node $T_{k,l}$ to the sink node t by an arc with capacity $(l - k + 1)M$, representing the total number of machine days available on the days from k to l. Finally, we connect a job node j to every interval node $T_{k,l}$ if $r_j \leq k$ and $d_j \geq l + 1$ by an arc with capacity $(l - k + 1)$ which represents the maximum number of machines days we can allot to job j on the days from k to l. We next solve a maximum flow problem on this network: The scheduling problem has a feasible schedule if and only if the maximum flow value equals $\sum_{j \in J} p_j$ [alternatively, the flow on every arc (s, j) is p_j]. The validity of this formulation is easy to establish by showing a one-to-one correspondence between feasible schedules and flows of value $\sum_{j \in J} p_j$ from the source to the sink.

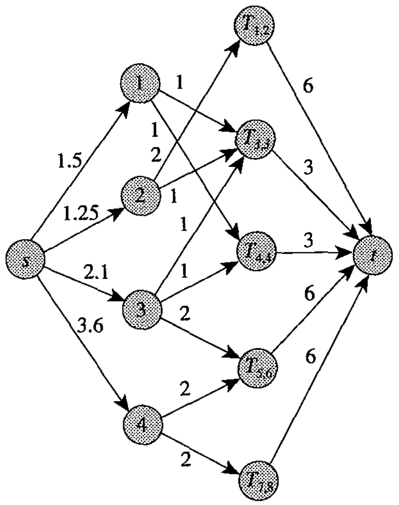


Figure 6.5 Network for scheduling uniform parallel machines.

Application 6.5 Distributed Computing on a Two-Processor Computer

This application concerns assigning different modules (subroutines) of a program to two processors in a way that minimizes the collective costs of interprocessor communication and computation. We consider a computer system with two processors; they need not be identical. We wish to execute a large program on this computer system. Each program contains several modules that interact with each other during the program's execution. The cost of executing each module on the two processes is known in advance and might vary from one processor to the other because of differences in the processors' memory, control, speed, and arithmetic capabilities. Let α_i and β_i denote the cost of computation of module i on processors 1 and 2, respectively. Assigning different modules to different processors incurs relatively high overhead costs due to interprocessor communication. Let c_{ij} denote the interprocessor communication cost if modules i and j are assigned to different processors; we do not incur this cost if we assign modules i and j to the same processor. The cost structure might suggest that we allocate two jobs to different processors—we need to balance this cost against the communication costs that we incur by allocating the jobs to different processors. Therefore, we wish to allocate modules of the program on the two processors so that we minimize the total cost of processing and interprocessor communication.

We formulate this problem as a minimum cut problem on an undirected network as follows. We define a source node s representing processor 1, a sink node t representing processor 2, and a node for every module of the program. For every node i, other than the source and sink nodes, we include an arc (s, i) of capacity β_i and an arc (i, t) of capacity α_i. Finally, if module i interacts with module j during program execution, we include the arc (i, j) with a capacity equal to c_{ij}. Figures 6.6 and 6.7 give an example of this construction. Figure 6.6 gives the data for this problem, and Figure 6.7 gives the corresponding network.

We now observe a one-to-one correspondence between s–t cuts in the network

i	1	2	3	4
α_i	6	5	10	4
β_i	4	10	3	8

(a)

$$\{c_{ij}\} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & 0 & 0 \\ 2 & 5 & 0 & 6 & 2 \\ 3 & 0 & 6 & 0 & 1 \\ 4 & 0 & 2 & 1 & 0 \end{array}$$

(b)

Figure 6.6 Data for the distributed computing model.

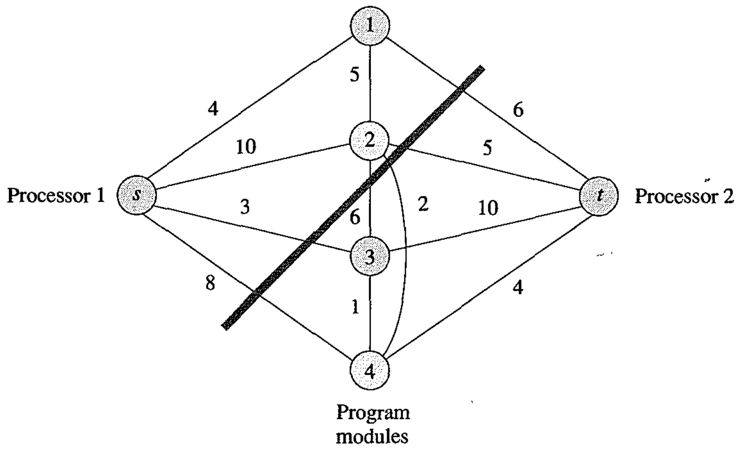


Figure 6.7 Network for the distributed computing model.

and assignments of modules to the two processors; moreover, the capacity of a cut equals the cost of the corresponding assignment. To establish this result, let A_1 and A_2 be an assignment of modules to processors 1 and 2, respectively. The cost of this assignment is $\sum_{i \in A_1} \alpha_i + \sum_{i \in A_2} \beta_i + \sum_{(i,j) \in A_1 x A_2} c_{ij}$. The s–t cut corresponding to this assignment is $(\{s\} \cup A_1, \{t\} \cup A_2)$. The approach we used to construct the network implies that this cut contains an arc (i, t) for every $i \in A_1$ of capacity α_i, an arc (s, i) for every $i \in A_2$ of capacity β_i, and all arcs (i, j) with $i \in A_1$ and $j \in A_2$ with capacity c_{ij}. The cost of the assignment A_1 and A_2 equals the capacity of the cut $(\{s\} \cup A_1, \{t\} \cup A_2)$. (We suggest that readers verify this conclusion using the example given in Figure 6.7 with $A_1 = \{1, 2\}$ and $A_2 = \{3, 4\}$.) Consequently, the minimum s–t cut in the network gives the minimum cost assignment of the modules to the two processors.

Application 6.6 Tanker Scheduling Problem

A steamship company has contracted to deliver perishable goods between several different origin–destination pairs. Since the cargo is perishable, the customers have specified precise dates (i.e., delivery dates) when the shipments must reach their destinations. (The cargoes may not arrive early or late.) The steamship company wants to determine the minimum number of ships needed to meet the delivery dates of the shiploads.

To illustrate a modeling approach for this problem, we consider an example with four shipments; each shipment is a full shipload with the characteristics shown in Figure 6.8(a). For example, as specified by the first row in this figure, the company must deliver one shipload available at port A and destined for port C on day 3. Figure 6.8(b) and (c) show the transit times for the shipments (including allowances for loading and unloading the ships) and the return times (without a cargo) between the ports.

Ship-ment	Origin	Desti-nation	Delivery date
1	Port A	Port C	3
2	Port A	Port C	8
3	Port B	Port D	3
4	Port B	Port C	6

(a)

	C	D
A	3	2
B	2	3

(b)

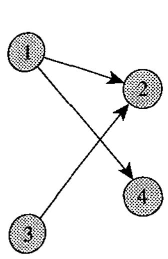
	A	B
C	2	1
D	1	2

(c)

Figure 6.8 Data for the tanker scheduling problem: (a) shipment characteristics; (b) shipment transit times; (c) return times.

We solve this problem by constructing a network shown in Figure 6.9(a). This network contains a node for each shipment and an arc from node i to node j if it is possible to deliver shipment j after completing shipment i; that is, the start time of shipment j is no earlier than the delivery time of shipment i plus the travel time from the destination of shipment i to the origin of shipment j. A directed path in this network corresponds to a feasible sequence of shipment pickups and deliveries. The tanker scheduling problem requires that we identify the minimum number of directed paths that will contain each node in the network on exactly one path.

We can transform this problem to the framework of the maximum flow problem as follows. We split each node i into two nodes i' and i'' and add the arc (i', i''). We set the lower bound on each arc (i', i''), called the *shipment arc*, equal to 1 so that at least one unit of flow passes through this arc. We also add a source node s and connect it to the origin of each shipment (to represent putting a ship into service),



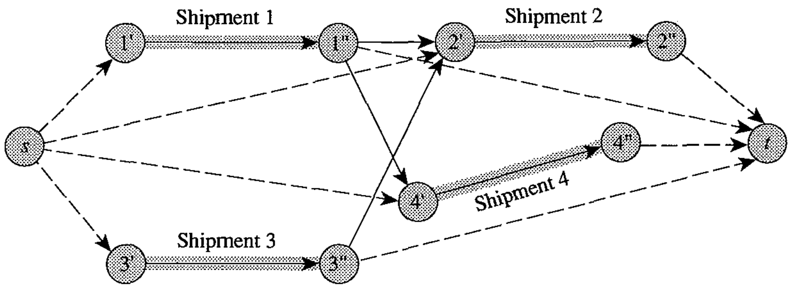


Figure 6.9 Network formulation of the tanker scheduling problem: (a) network of feasible sequences of two consecutive shipments; (b) maximum flow model.

and we add a sink node t and connect each destination node to it (to represent taking a ship out of service). We set the capacity of each arc in the network to value 1. Figure 6.9(b) shows the resulting network for our example. In this network, each directed path from the source s to the sink t corresponds to a feasible schedule for a single ship. As a result, a feasible flow of value v in this network decomposes into schedules of v ships and our problem reduces to identifying a feasible flow of minimum value. We note that the zero flow is not feasible because shipment arcs have unit lower bounds. We can solve this problem, which is known as the *minimum value problem*, using any maximum flow algorithm (see Exercise 6.18).

6.3 FLOWS AND CUTS

In this section we discuss some elementary properties of flows and cuts. We use these properties to prove the max-flow min-cut theorem to establish the correctness of the generic augmenting path algorithm. We first review some of our previous notation and introduce a few new ideas.

Residual network. The concept of *residual network* plays a central role in the development of all the maximum flow algorithms we consider. Earlier in Section 2.4 we defined residual networks and discussed several of its properties. Given a flow x, the residual capacity r_{ij} of any arc $(i, j) \in A$ is the maximum additional flow that can be sent from node i to node j using the arcs (i, j) and (j, i). [Recall our assumption from Section 6.1 that whenever the network contains arc (i, j), it also contains arc (j, i).] The residual capacity r_{ij} has two components: (1) $u_{ij} - x_{ij}$, the unused capacity of arc (i, j), and (2) the current flow x_{ji} on arc (j, i), which we can cancel to increase the flow from node i to node j. Consequently, $r_{ij} = u_{ij} - x_{ij} + x_{ji}$. We refer to the network $G(x)$ consisting of the arcs with positive residual capacities as the *residual network* (with respect to the flow x). Figure 6.10 gives an example of a residual network.

***s–t* cut.** We now review notation about cuts. Recall from Section 2.2 that a cut is a partition of the node set N into two subsets S and $\bar{S} = N - S$; we represent this cut using the notation $[S, \bar{S}]$. Alternatively, we can define a cut as the set of

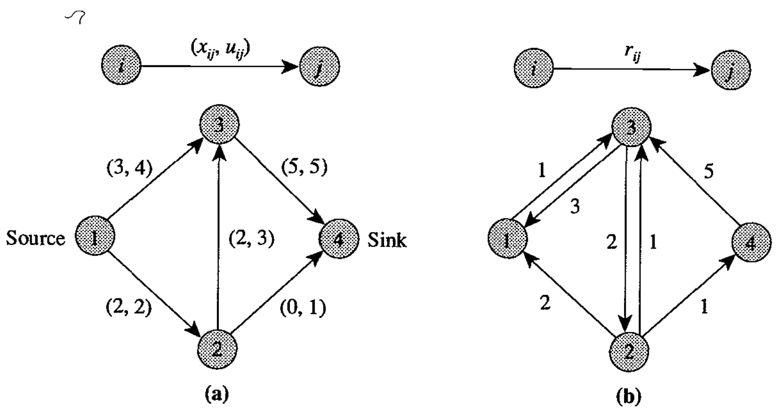


Figure 6.10 Illustrating a residual network: (a) original network G with a flow x; (b) residual network $G(x)$.

arcs whose endpoints belong to the different subsets S and $\bar{S}$. We refer to a cut as an *s–t cut* if $s \in S$ and $t \in \bar{S}$. We also refer to an arc (i, j) with $i \in S$ and $j \in \bar{S}$ as a *forward arc* of the cut, and an arc (i, j) with $i \in \bar{S}$ and $j \in S$ as a *backward arc* of the cut $[S, \bar{S}]$. Let $(S, \bar{S})$ denote the set of forward arcs in the cut, and let $(\bar{S}, S)$ denote the set of backward arcs. For example, in Figure 6.11, the dashed arcs constitute an s–t cut. For this cut, $(S, \bar{S}) = \{(1, 2), (3, 4), (5, 6)\}$, and $(\bar{S}, S) = \{(2, 3), (4, 5)\}$.

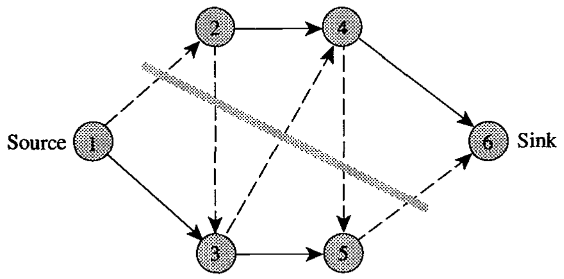


Figure 6.11 Example of an s–t cut.

Capacity of an s–t cut. We define the capacity $u[S, \bar{S}]$ of an s–t cut $[S, \bar{S}]$ as the sum of the capacities of the forward arcs in the cut. That is,

$$u[S, \bar{S}] = \sum_{(i,j)\in(S,\bar{S})} u_{ij}.$$

Clearly, the capacity of a cut is an upper bound on the maximum amount of flow we can send from the nodes in S to the nodes in $\bar{S}$ while honoring arc flow bounds.

Minimum cut. We refer to an s–t cut whose capacity is minimum among all s–t cuts as a *minimum cut.*

Residual capacity of an s–t cut. We define the residual capacity $r[S, \bar{S}]$ of an s–t cut $[S, \bar{S}]$ as the sum of the residual capacities of forward arcs in the cut. That is,

$$r[S, \bar{S}] = \sum_{(i,j) \in (S,\bar{S})} r_{ij}.$$

Flow across an *s–t* cut. Let x be a flow in the network. Adding the mass balance constraint (6.1b) for the nodes in S, we see that

$$v = \sum_{i \in S} \left[\sum_{\{j:(i,j) \in A\}} x_{ij} - \sum_{\{j:(j,i) \in A\}} x_{ji} \right].$$

We can simplify this expression by noting that whenever both the nodes p and q belong to S and $(p, q) \in A$, the variable x_{pq} in the first term within the brackets (for node $i = p$) cancels the variable $-x_{pq}$ in the second term within the brackets (for node $j = q$). Moreover, if both the nodes p and q belong to $\bar{S}$, then x_{pq} does not appear in the expression. This observation implies that

$$v = \sum_{(i,j) \in (S,\bar{S})} x_{ij} - \sum_{(i,j) \in (\bar{S},S)} x_{ij}. \tag{6.3}$$

The first expression on the right-hand side of (6.3) denotes the amount of flow from the nodes in S to nodes in $\bar{S}$, and the second expression denotes the amount of flow returning from the nodes in $\bar{S}$ to the nodes in S. Therefore, the right-hand side denotes the total (net) flow across the cut, and (6.3) implies that the flow across *any s–t* cut $[S, \bar{S}]$ equals v. Substituting $x_{ij} \le u_{ij}$ in the first expression of (6.3) and $x_{ij} \ge 0$ in the second expression shows that

$$v \le \sum_{(i,j) \in (S,\bar{S})} u_{ij} = u[S, \bar{S}]. \tag{6.4}$$

This expression indicates that the value of *any* flow is less than or equal to the capacity of *any s–t* cut in the network. This result is also quite intuitive. Any flow from node s to node t must pass through every $s–t$ cut in the network (because any cut divides the network into two disjoint components), and therefore the value of the flow can never exceed the capacity of the cut. Let us formally record this result.

Property 6.1. *The value of any flow is less than or equal to the capacity of any cut in the network.*

This property implies that if we discover a flow x whose value equals the capacity of some cut $[S, \bar{S}]$, then x is a maximum flow and the cut $[S, \bar{S}]$ is a minimum cut. The max-flow min-cut theorem, proved in the next section, states that some flow always has a flow value equal to the capacity of some cut.

We next restate Property 6.1 in terms of the residual capacities. Suppose that x is a flow of value v. Moreover, suppose that that x' is a flow of value $v + \Delta v$ for some $\Delta v \ge 0$. The inequality (6.4) implies that

$$v + \Delta v \le \sum_{(i,j) \in (S,\bar{S})} u_{ij}. \tag{6.5}$$

Subtracting (6.3) from (6.5) shows that

$$\Delta v \le \sum_{(i,j) \in (S,\bar{S})} (u_{ij} - x_{ij}) + \sum_{(i,j) \in (\bar{S},S)} x_{ij}. \tag{6.6}$$

We now use Assumption 6.4 to note that we can rewrite $\sum_{(i,j)\in(\bar{S},S)} x_{ij}$ as $\sum_{(i,j)\in(S,\bar{S})} x_{ji}$. Consequently,

$$\Delta v \leq \sum_{(i,j)\in(S,\bar{S})} (u_{ij} - x_{ij} + x_{ji}) = \sum_{(S,\bar{S})} r_{ij}.$$

The following property is now immediate.

Property 6.2. *For any flow x of value v in a network, the additional flow that can be sent from the source node s to the sink node t is less than or equal to the residual capacity of any s–t cut.*

6.4 GENERIC AUGMENTING PATH ALGORITHM

In this section, we describe one of the simplest and most intuitive algorithms for solving the maximum flow problem. This algorithm is known as the *augmenting path algorithm.*

We refer to a directed path from the source to the sink in the residual network as an *augmenting path.* We define the residual capacity of an augmenting path as the minimum residual capacity of any arc in the path. For example, the residual network in Figure 6.10(b), contains exactly one augmenting path 1–3–2–4, and the residual capacity of this path is $\delta = \min\{r_{13}, r_{32}, r_{24}\} = \min\{1, 2, 1\} = 1$. Observe that, by definition, the capacity δ of an augmenting path is always positive. Consequently, whenever the network contains an augmenting path, we can send additional flow from the source to the sink. The generic augmenting path algorithm is essentially based on this simple observation. The algorithm proceeds by identifying augmenting paths and augmenting flows on these paths until the network contains no such path. Figure 6.12 describes the generic augmenting path algorithm.

```
algorithm augmenting path;
begin
    $x := 0$;
    while $G(x)$ contains a directed path from node $s$ to node $t$ do
    begin
        identify an augmenting path $P$ from node $s$ to node $t$;
        $\delta := \min\{r_{ij} : (i, j) \in P\}$;
        augment $\delta$ units of flow along $P$ and update $G(x)$;
    end;
end;
```

Figure 6.12 Generic augmenting path algorithm.

We use the maximum flow problem given in Figure 6.13(a) to illustrate the algorithm. Suppose that the algorithm selects the path 1–3–4 for augmentation. The residual capacity of this path is $\delta = \min\{r_{13}, r_{34}\} = \min\{4, 5\} = 4$. This augmentation reduces the residual capacity of arc (1, 3) to zero (thus we delete it from the residual network) and increases the residual capacity of arc (3, 1) to 4 (so we add this arc to the residual network). The augmentation also decreases the residual capacity of arc (3, 4) from 5 to 1 and increases the residual capacity of arc (4, 3) from 0 to 4. Figure 6.13(b) shows the residual network at this stage. In the second iteration, suppose that the algorithm selects the path 1–2–3–4. The residual capacity of this

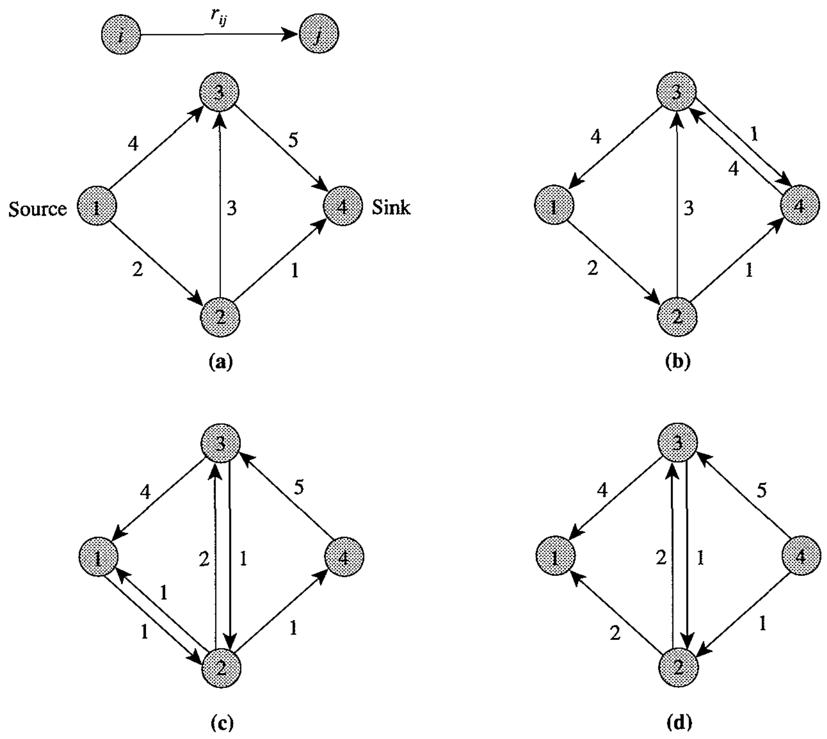


Figure 6.13 Illustrating the generic augmenting path algorithm: (a) residual network for the zero flow; (b) network after augmenting four units along the path 1–3–4; (c) network after augmenting one unit along the path 1–2–3–4; (d) network after augmenting one unit along the path 1–2–4.

path is $\delta = \min\{2, 3, 1\} = 1$. Augmenting 1 unit of flow along this path yields the residual network shown in Figure 6.13(c). In the third iteration, the algorithm augments 1 unit of flow along the path 1–2–4. Figure 6.13(d) shows the corresponding residual network. Now the residual network contains no augmenting path, so the algorithm terminates.

Relationship between the Original and Residual Networks

In implementing any version of the generic augmenting path algorithm, we have the option of working directly on the original network with the flows x_{ij}, or maintaining the residual network $G(x)$ and keeping track of the residual capacities r_{ij} and, when the algorithm terminates, recovering the actual flow variables x_{ij}. To see how we can use either alternative, it is helpful to understand the relationship between arc flows in the original network and residual capacities in the residual network.

First, let us consider the concept of an augmenting path in the original network. An augmenting path in the original network G is a path P (not necessarily directed) from the source to the sink with $x_{ij} < u_{ij}$ on every forward arc (i, j) and $x_{ij} > 0$ on every backward arc (i, j). It is easy to show that the original network G contains an augmenting path with respect to a flow x if and only if the residual network $G(x)$ contains a directed path from the source to the sink.

Now suppose that we update the residual capacities at some point in the algorithm. What is the effect on the arc flows x_{ij}? The definition of the residual capacity (i.e., $r_{ij} = u_{ij} - x_{ij} + x_{ji}$) implies that an additional flow of δ units on arc (i, j) in the residual network corresponds to (1) an increase in x_{ij} by δ units in the original network, or (2) a decrease in x_{ji} by δ units in the original network, or (3) a convex combination of (1) and (2). We use the example given in Figure 6.14(a) and the corresponding residual network in Figure 6.14(b) to illustrate these possibilities. Augmenting 1 unit of flow on the path 1–2–4–3–5–6 in the network produces the residual network in Figure 6.14(c) with the corresponding arc flows shown in Figure 6.14(d). Comparing the solution in Figure 6.14(d) with that in Figure 6.14(a), we find that the flow augmentation increases the flow on arcs (1, 2), (2, 4), (3, 5), (5, 6) and decreases the flow on arc (3, 4).

Finally, suppose that we are given values for the residual capacities. How should we determine the flows x_{ij}? Observe that since $r_{ij} = u_{ij} - x_{ij} + x_{ji}$, many combinations of x_{ij} and x_{ji} correspond to the same value of r_{ij}. We can determine

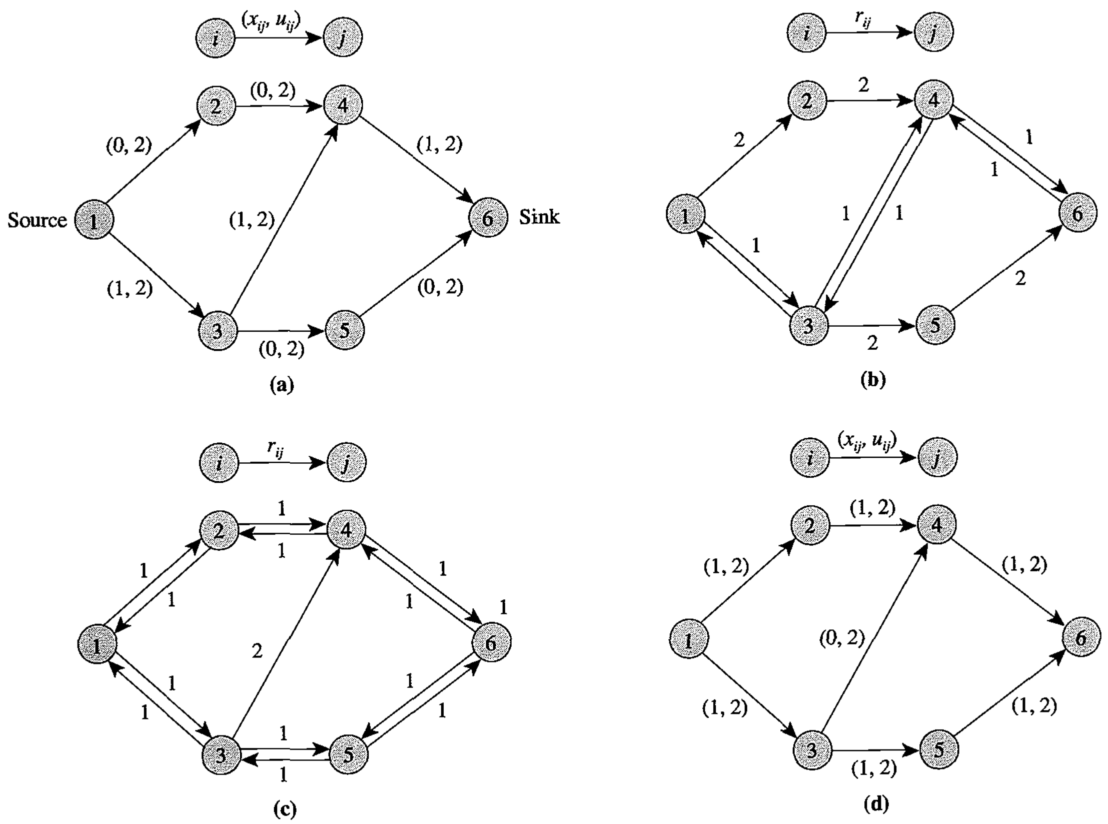


Figure 6.14 The effect of augmentation on flow decomposition: (a) original network with a flow x; (b) residual network for flow x; (c) residual network after augmenting one unit along the path 1–2–4–3–5–6; (d) flow in the original network after the augmentation.

one such choice as follows. To highlight this choice, let us rewrite $r_{ij} = u_{ij} - x_{ij} + x_{ji}$ as $x_{ij} - x_{ji} = u_{ij} - r_{ij}$. Now, if $u_{ij} \geq r_{ij}$, we set $x_{ij} = u_{ij} - r_{ij}$ and $x_{ji} = 0$; otherwise, we set $x_{ij} = 0$ and $x_{ji} = r_{ij} - u_{ij}$.

Effect of Augmentation on Flow Decomposition

To obtain better insight concerning the augmenting path algorithm, let us illustrate the effect of an augmentation on the flow decomposition on the preceding example. Figure 6.15(a) gives the decomposition of the initial flow and Figure 6.15(b) gives the decomposition of the flow after we have augmented 1 unit of flow on the path 1–2–4–3–5–6. Although we augmented 1 unit of flow along the path 1–2–4–3–5–6, the flow decomposition contains no such path. Why?

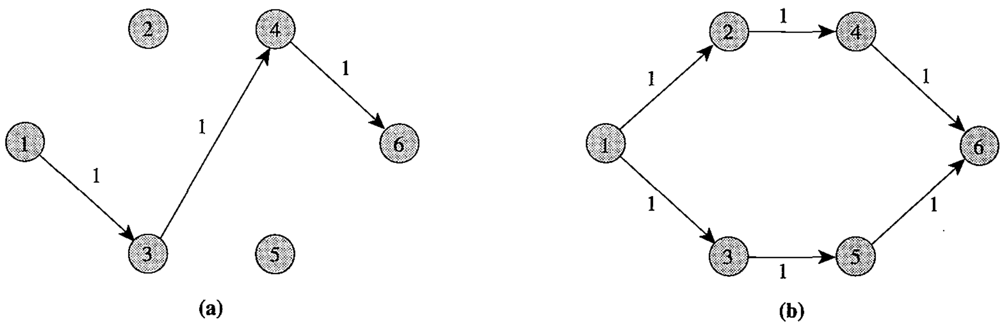


Figure 6.15 Flow decomposition of the solution in (a) Figure 6.14(a) and (b) Figure 6.14(d).

The path 1–3–4–6 defining the flow in Figure 6.14(a) contains three segments: the path up to node 3, arc (3, 4) as a forward arc, and the path up to node 6. We can view this path as an augmentation on the zero flow. Similarly, the path 1–2–4–3–5–6 contains three segments: the path up to node 4, arc (3, 4) as a backward arc, and the path up to node 6. We can view the augmentation on the path 1–2–4–3–5–6 as linking the initial segment of the path 1–3–4–6 with the last segment of the augmentation, linking the last segment of the path 1–3–4–6 with the initial segment of the augmentation, and canceling the flow on arc (3, 4), which then drops from both the path 1–3–4–6 and the augmentation (see Figure 6.16). In general, we can

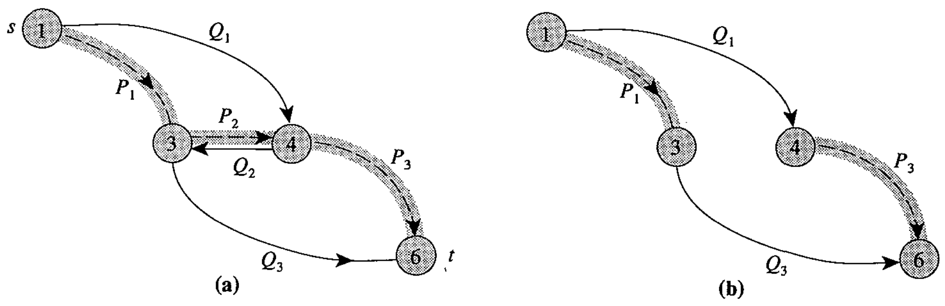


Figure 6.16 The effect of augmentation on flow decomposition: (a) the two augmentations P_1–P_2–P_3 and Q_1–Q_2–Q_3; (b) net effect of these augmentations.

view each augmentation as "pasting together" segments of the current flow decomposition to obtain a new flow decomposition.

6.5 *LABELING ALGORITHM AND THE MAX-FLOW MIN-CUT THEOREM*

In this section we discuss the augmenting path algorithm in more detail. In our discussion of this algorithm in the preceding section, we did not discuss some important details, such as (1) how to identify an augmenting path or show that the network contains no such path, and (2) whether the algorithm terminates in finite number of iterations, and when it terminates, whether it has obtained a maximum flow. In this section we consider these issues for a specific implementation of the generic augmenting path algorithm known as the *labeling algorithm*. The labeling algorithm is not a polynomial-time algorithm. In Chapter 7, building on the ideas established in this chapter, we describe two polynomial-time implementations of this algorithm.

The labeling algorithm uses a search technique (as described in Section 3.4) to identify a directed path in $G(x)$ from the source to the sink. The algorithm *fans out* from the source node to find all nodes that are reachable from the source along a directed path in the residual network. At any step the algorithm has partitioned the nodes in the network into two groups: *labeled* and *unlabeled*. Labeled nodes are those nodes that the algorithm has reached in the fanning out process and so the algorithm has determined a directed path from the source to these nodes in the residual network; the unlabeled nodes are those nodes that the algorithm has not reached as yet by the fanning-out process. The algorithm iteratively selects a labeled node and scans its arc adjacency list (in the residual network) to reach and label additional nodes. Eventually, the sink becomes labeled and the algorithm sends the maximum possible flow on the path from node s to node t. It then erases the labels and repeats this process. The algorithm terminates when it has scanned all the labeled nodes and the sink remains unlabeled, implying that the source node is not connected to the sink node in the residual network. Figure 6.17 gives an algorithmic description of the labeling algorithm.

Correctness of the Labeling Algorithm and Related Results

To study the correctness of the labeling algorithm, note that in each iteration (i.e., an execution of the whole loop), the algorithm either performs an augmentation or terminates because it cannot label the sink. In the latter case we must show that the current flow x is a maximum flow. Suppose at this stage that S is the set of labeled nodes and $\bar{S} = N - S$ is the set of unlabeled nodes. Clearly, $s \in S$ and $t \in \bar{S}$. Since the algorithm cannot label any node in $\bar{S}$ from any node in S, $r_{ij} = 0$ for each $(i, j) \in (S, \bar{S})$. Furthermore, since $r_{ij} = (u_{ij} - x_{ij}) + x_{ji}$, $x_{ij} \le u_{ij}$ and $x_{ji} \ge 0$, the condition $r_{ij} = 0$ implies that $x_{ij} = u_{ij}$ for every arc $(i, j) \in (S, \bar{S})$ and $x_{ij} = 0$ for every arc $(i, j) \in (\bar{S}, S)$. [Recall our assumption that for each arc $(i, j) \in A$,

```
algorithm labeling;
begin
    label node $t$;
    while $t$ is labeled do
    begin
        unlabel all nodes;
        set pred($j$) : = 0 for each $j \in N$;
        label node $s$ and set LIST : = $\{s\}$;
        while LIST $\neq \emptyset$ or $t$ is unlabeled do
        begin
            remove a node $i$ from LIST;
            for each arc $(i, j)$ in the residual network emanating from node $i$ do
                if $r_{ij} > 0$ and node $j$ is unlabeled then set pred($j$) : = $i$, label node $j$, and
                    add $j$ to LIST;
        end;
        if $t$ is labeled then augment
    end;
end;

procedure augment;
begin
    use the predecessor labels to trace back from the sink to the source to
        obtain an augmenting path $P$ from node $s$ to node $t$;
    $\delta$ : = min$\{r_{ij} : (i, j) \in P\}$;
    augment $\delta$ units of flow along $P$ and update the residual capacities;
end;
```

Figure 6.17 Labeling algorithm.

$(j, i) \in A$.] Substituting these flow values in (6.3), we find that

$$v = \sum_{(i,j)\in(S,\bar{S})} x_{ij} - \sum_{(i,j)\in(\bar{S},S)} x_{ij} = \sum_{(i,j)\in(S,\bar{S})} u_{ij} = u[S, \bar{S}].$$

This discussion shows that the value of the current flow x equals the capacity of the cut $[S, \bar{S}]$. But then Property 6.1 implies that x is a maximum flow and $[S, \bar{S}]$ is a minimum cut. This conclusion establishes the correctness of the labeling algorithm and, as a by-product, proves the following max-flow min-cut theorem.

Theorem 6.3 (Max-Flow Min-Cut Theorem). *The maximum value of the flow from a source node s to a sink node t in a capacitated network equals the minimum capacity among all s–t cuts.* ◆

The proof of the max-flow min-cut theorem shows that when the labeling algorithm terminates, it has also discovered a minimum cut. The labeling algorithm also proves the following augmenting path theorem.

Theorem 6.4 (Augmenting Path Theorem). *A flow x^* is a maximum flow if and only if the residual network $G(x^*)$ contains no augmenting path.*

Proof. If the residual network $G(x^*)$ contains an augmenting path, clearly the flow x^* is not a maximum flow. Conversely, if the residual network $G(x^*)$ contains no augmenting path, the set of nodes S labeled by the labeling algorithm defines an

s–t cut $[S, \bar{S}]$ whose capacity equals the flow value, thereby implying that the flow must be maximum. ◆

The labeling algorithm establishes one more important result.

Theorem 6.5 (Integrality Theorem). *If all arc capacities are integer, the maximum flow problem has an integer maximum flow.*

Proof. This result follows from an induction argument applied to the number of augmentations. Since the labeling algorithm starts with a zero flow and all arc capacities are integer, the initial residual capacities are all integer. The flow augmented in any iteration equals the minimum residual capacity of some path, which by the induction hypothesis is integer. Consequently, the residual capacities in the next iteration will again be integer. Since the residual capacities r_{ij} and the arc capacities u_{ij} are all integer, when we convert the residual capacities into flows by the method described previously, the arc flows x_{ij} will be integer valued as well. Since the capacities are integer, each augmentation adds at least one unit to the flow value. Since the maximum flow cannot exceed the capacity of any cut, the algorithm will terminate in a finite number of iterations. ◆

The integrality theorem does not imply that every optimal solution of the maximum flow problem is integer. The maximum flow problem may have noninteger solutions and, most often, has such solutions. The integrality theorem shows that the problem always has at least one integer optimal solution.

Complexity of the Labeling Algorithm

To study the worst-case complexity of the labeling algorithm, recall that in each iteration, except the last, when the sink cannot be labeled, the algorithm performs an augmentation. It is easy to see that each augmentation requires $O(m)$ time because the search method examines any arc or any node at most once. Therefore, the complexity of the labeling algorithm is $O(m)$ times the number of augmentations. How many augmentations can the algorithm perform? If all arc capacities are integral and bounded by a finite number U, the capacity of the cut $(s, N - \{s\})$ is at most nU. Therefore, the maximum flow value is bounded by nU. The labeling algorithm increases the value of the flow by at least 1 unit in any augmentation. Consequently, it will terminate within nU augmentations, so $O(nmU)$ is a bound on the running time of the labeling algorithm. Let us formally record this observation.

Theorem 6.6. *The labeling algorithm solves the maximum flow problem in $O(nmU)$ time.* ◆

Throughout this section, we have assumed that each arc capacity is finite. In some applications, it will be convenient to model problems with infinite capacities on some arcs. If we assume that some s–t cut has a finite capacity and let U denote the maximum capacity across this cut, Theorem 6.6 and, indeed, all the other results in this section remain valid. Another approach for addressing situations with infinite capacity arcs would be to impose a capacity on these arcs, chosen sufficiently large as to not affect the maximum flow value (see Exercise 6.23). In defining the residual capacities and developing algorithms to handle situations with infinite arc capacities, we adopt this approach rather than modifying the definitions of residual capacities.

Drawbacks of the Labeling Algorithm

The labeling algorithm is possibly the simplest algorithm for solving the maximum flow problem. Empirically, the algorithm performs reasonably well. However, the worst-case bound on the number of iterations is not entirely satisfactory for large values of U. For example, if $U = 2^n$, the bound is exponential in the number of nodes. Moreover, the algorithm can indeed perform this many iterations, as the example given in Figure 6.18 illustrates. For this example, the algorithm can select the augmenting paths s–a–b–t and s–b–a–t alternatively 10^6 times, each time augmenting unit flow along the path. This example illustrates one shortcoming of the algorithm.

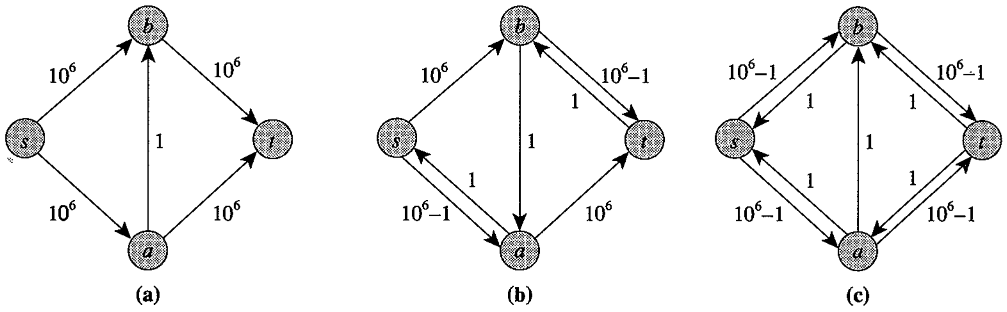


Figure 6.18 Pathological example of the labeling algorithm: (a) residual network for the zero flow; (b) network after augmenting unit flow along the path s–a–b–t; (c) network after augmenting unit flow along the path s–b–a–t.

A second drawback of the labeling algorithm is that if the capacities are irrational, the algorithm might not terminate. For some pathological instances of the maximum flow problem (see Exercise 6.48), the labeling algorithm does not terminate, and although the successive flow values converge, they converge to a value strictly less than the maximum flow value. (Note, however, that the max-flow min-cut theorem holds even if arc capacities are irrational.) Therefore, if the labeling algorithm is guaranteed to be effective, it must select augmenting paths carefully.

A third drawback of the labeling algorithm is its *"forgetfulness."* In each iteration, the algorithm generates node labels that contain information about augmenting paths from the source to other nodes. The implementation we have described erases the labels as it moves from one iteration to the next, even though much of this information might be valid in the next iteration. Erasing the labels therefore destroys potentially useful information. Ideally, we should retain a label when we can use it profitably in later computations.

In Chapter 7 we describe several improvements of the labeling algorithm that overcomes some or all of these drawbacks. Before discussing these improvements, we discuss some interesting implications of the max-flow min-cut theorem.

6.6 COMBINATORIAL IMPLICATIONS OF THE MAX-FLOW MIN-CUT THEOREM

As we noted in Section 6.2 when we discussed several applications of the maximum flow problem, in some applications we wish to find a minimum cut in a network, which we now know is equivalent to finding a maximum flow in the network. In fact, the relationship between maximum flows and minimum cuts permits us to view many problems from either of two dual perspectives: a flow perspective or a cut perspective. At times this dual perspective provides novel insight about an underlying problem. In particular, when applied in various ways, the max-flow min-cut theorem reduces to a number of min-max duality relationships in combinatorial theory. In this section we illustrate this use of network flow theory by developing several results in combinatorics. We might note that these results are fairly deep and demonstrate the power of the max-flow min-cut theorem. To appreciate the power of the max-flow min-cut theorem, we would encourage the reader to try to prove the following results without using network flow theory.

Network Connectivity

We first study some connectivity issues about networks that arise, for example, in the design of communication networks. We first define some notation. We refer to two directed paths from node s to node t as *arc disjoint* if they do not have any arc in common. Similarly, we refer to two directed paths from node s to node t as *node disjoint* if they do not have any node in common, except the source and the sink nodes. Given a directed network $G = (N, A)$ and two specified nodes s and t, we are interested in the following two questions: (1) What is the maximum number of arc-disjoint (directed) paths from node s to node t; and (2) what is the minimum number of arcs that we should remove from the network so that it contains no directed paths from node s to node t? The following theorem shows that these two questions are really alternative ways to address the same issue.

Theorem 6.7. *The maximum number of arc-disjoint paths from node s to node t equals the minimum number of arcs whose removal from the network disconnects all paths from node s to node t.*

Proof. Define the capacity of each arc in the network as equal to 1. Consider any feasible flow x of value v in the resulting unit capacity network. The flow decomposition theorem (Theorem 3.5) implies that we can decompose the flow x into flows along paths and cycles. Since flows around cycles do not affect the flow value, the flows on the paths sum to v. Furthermore, since each arc capacity is 1, these paths are arc disjoint and each carries 1 unit of flow. Consequently, the network contains v arc-disjoint paths from s to t.

Now consider any s–t cut $[S, \bar{S}]$ in the network. Since each arc capacity is 1, the capacity of this cut is $|(S, \bar{S})|$ (i.e., it equals the number of forward arcs in the cut). Since each path from node s to node t contains at least one arc in $(S, \bar{S})$, the removal of the arcs in $(S, \bar{S})$ disconnects all the paths from node s to node t. Consequently, the network contains a disconnecting set of arcs of cardinality equal to the capacity of any s–t cut $[S, \bar{S}]$. The max-flow min-cut theorem immediately implies that the maximum number of arc-disjoint paths from s to t equals the minimum number of arcs whose removal will disconnect all paths from node s to node t.

◆

We next discuss the node-disjoint version of the preceding theorem.

Theorem 6.8. *The maximum number of node-disjoint paths from node s to node t equals the minimum number of nodes whose removal from the network disconnects all paths from nodes s to node t.*

Proof. Split each node i in G, other than s and t, into two nodes i' and i'' and add a "node-splitting" arc (i', i'') of unit capacity. All the arcs in G entering node i now enter node i' and all the arcs emanating from node i now emanate from node i''. Let G' denote this transformed network. Assign a capacity of ∞ to each arc in the network except the node-splitting arcs, which have unit capacity. It is easy to see that there is one-to-one correspondence between the arc-disjoint paths in G' and the node-disjoint paths in G. Therefore, the maximum number of arc-disjoint paths in G' equals the maximum number of node-disjoint paths in G.

As in the proof of Theorem 6.7, flow decomposition implies that a flow of v units from node s to node t in G' decomposes into v arc-disjoint paths each carrying unit flow; and these v arc-disjoint paths in G' correspond to v node-disjoint paths in G. Moreover, note that any s–t cut with finite capacity contains only node-splitting arcs since all other arcs have infinite capacity. Therefore, any s–t cut in G' with capacity k corresponds to a set of k nodes whose removal from G destroys all paths from node s to node t. Applying the max-flow min-cut theorem to G' and using the preceding observations establishes that the maximum number of node-disjoint paths in G from node s to node t equals the minimum number of nodes whose removal from G disconnects nodes s and t. ◆

Matchings and Covers

We next state some results about matchings and node covers in a bipartite network. For a directed bipartite network $G = (N_1 \cup N_2, A)$ we refer to a subset $A' \subseteq A$ as a *matching* if no two arcs in A' are incident to the same node (i.e., they do not have any common endpoint). We refer to a subset $N' \subseteq N = N_1 \cup N_2$ as a *node cover* if every arc in A is incident to one of the nodes in N'. For illustrations of these definitions, consider the bipartite network shown in Figure 6.19. In this network the set of arcs $\{(1, 1'), (3, 3'), (4, 5'), (5, 2')\}$ is a matching but the set of arcs $\{(1, 2'), (3, 1'), (3, 4')\}$ is not because the arcs $(3, 1')$ and $(3, 4')$ are incident to the same node 3. In the same network the set of nodes $\{1, 2', 3, 5'\}$ is a node cover, but the set of nodes $\{2', 3', 4, 5\}$ is not because the arcs $(1, 1')$, $(3, 1')$, and $(3, 4')$ are not incident to any node in the set.

Theorem 6.9. *In a bipartite network $G = (N_1 \cup N_2, A)$, the maximum cardinality of any matching equals the minimum cardinality of any node cover of G.*

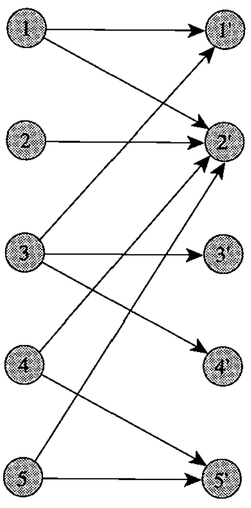


Figure 6.19 Bipartite network.

Proof. Augment the network by adding a source node s and an arc (s, i) of unit capacity for each $i \in N_1$. Similarly, add a sink node t and an arc (j, t) of unit capacity for each $j \in N_2$. Denote the resulting network by G'. We refer to the arcs in A as *original arcs* and the additional arcs as *artificial arcs*. We set the capacity of each artificial arc equal to 1 and the capacity of each original arc equal to ∞.

Now consider any flow x of value v from node s to node t in the network G'. We can decompose the flow x into v paths of the form s–i–j–t each carrying 1 unit of flow. Thus v arcs of the original network have a positive flow. Furthermore, these arcs constitute a matching, for otherwise the flow on some artificial arc would exceed 1 unit. Consequently, a flow of value v corresponds to a matching of cardinality v. Similarly, a matching of cardinality v defines a flow of value v.

We next show that any node cover H of $G = (N_1 \cup N_2, A)$ defines an s–t cut of capacity $|H|$ in G'. Given the node cover H, construct a set of arcs Q as follows: For each $i \in H$, if $i \in N_1$, add arc (s, i) to Q, and if $i \in N_2$, add arc (i, t) to Q. Since H is a node cover, each directed path from node s to node t in G' contains one arc in Q; therefore, Q is a valid s–t cut of capacity $|H|$.

We now show the converse result; that is, for a given s–t cut Q of capacity k in G', the network G contains a node cover of cardinality k. We first note that the cut Q consists solely of artificial arcs because the original arcs have infinite capacity. From Q we construct a set H of nodes as follows: if $(s, i) \in Q$ and $(i, t) \in Q$, we add i to H. Now observe that each original arc (i, j) defines a directed path s–i–j–t in G'. Since Q is an s–t cut, either $(s, i) \in Q$ or $(j, t) \in Q$ or both. By the preceding construction, either $i \in H$ or $j \in H$ or both. Consequently, H must be a node cover. We have thus established a one-to-one correspondence between node covers in G and s–t cuts in G'.

The max-flow min-cut theorem implies that the maximum flow value equals the capacity of a minimum cut. In view of the max-flow min-cut theorem, the preceding observations imply that the maximum number of independent arcs in G equals the minimum number of nodes in a node cover of G. The theorem thus follows. ◆

Figure 6.20 gives a further illustration of Theorem 6.9. In this figure, we have transformed the matching problem of Figure 6.19 into a maximum flow problem, and we have identified the minimum cut. The minimum cut consists of the arcs $(s, 1)$, $(s, 3)$, $(2', t)$ and $(5', t)$. Correspondingly, the set $\{1, 3, 2', 5'\}$ is a minimum cardinality node cover, and a maximum cardinality matching is $(1, 1')$, $(2, 2')$, $(3, 3')$ and $(5, 5')$.

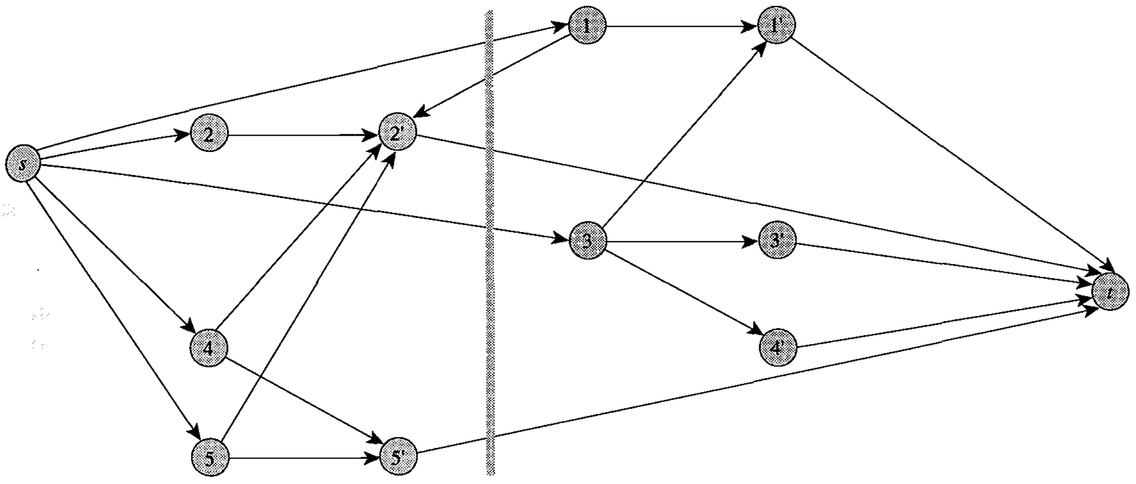


Figure 6.20 Minimum cut for the maximum flow problem defined in Figure 6.19.

As we have seen in the discussion throughout this section, the max-flow min-cut theorem is a powerful tool for establishing a number of results in the field of combinatorics. Indeed, the range of applications of the max-flow min-cut theorem and the ability of this theorem to encapsulate so many subtle duality (i.e., max-min) results as special cases is quite surprising, given the simplicity of the labeling algorithm and of the proof of the max-flow min-cut theorem. The wide range of applications reflects the fact that flows and cuts, and the relationship between them, embody central combinatorial results in many problem domains within applied mathematics.

6.7 FLOWS WITH LOWER BOUNDS

In this section we consider maximum flow problems with nonnegative lower bounds imposed on the arc flows; that is, the flow on any arc $(i, j) \in A$ must be at least $l_{ij} \geq 0$. The following formulation models this problem:

Maximize v

subject to

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

$$l_{ij} \leq x_{ij} \leq u_{ij} \qquad \text{for each } (i, j) \in A.$$

In previous sections we studied a special case of this problem with only zero lower bounds. Whereas the maximum flow problem with zero lower bounds always has a feasible solution (since the zero flow is feasible), the problem with nonnegative lower bounds could be infeasible. For example, consider the maximum flow problem given in Figure 6.21. This problem does not have a feasible solution because arc (1, 2) must carry at least 5 units of flow into node 2 and arc (2, 3) can remove at most 4 units of flow; therefore, we can never satisfy the mass balance constraint of node 2.

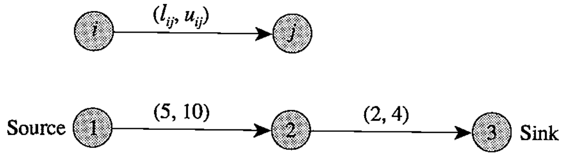


Figure 6.21 Maximum flow problem with no feasible solution.

As illustrated by this example, any maximum flow algorithm for problems with nonnegative lower bounds has two objectives: (1) to determine whether the problem is feasible, and (2) if so, to establish a maximum flow. It therefore comes as no surprise that most algorithms use a two-phase approach. The first phase determines a feasible flow if one exists, and the second phase converts a feasible flow into a maximum flow. We shall soon see that the problem in each phase essentially reduces to solving a maximum flow problem with zero lower bounds. Consequently, it is possible to solve the maximum flow problem with nonnegative lower bounds by solving two maximum flow problems, each with zero lower bounds. For convenience, we consider the second phase prior to the first phase.

Determining a Maximum Flow

Suppose that we have a feasible flow x in the network. We can then modify any maximum flow algorithm designed for the zero lower bound case to obtain a maximum flow. In these algorithms, we make only one modification: We define the residual capacity of an arc (i, j) as $r_{ij} = (u_{ij} - x_{ij}) + (x_{ji} - l_{ji})$; the first term in this expression denotes the maximum increase in flow from node i to node j using the remaining capacity of arc (i, j), and the second term denotes the maximum increase in flow from node i to node j by canceling the existing flow on arc (j, i). Notice that since each arc flow is within its lower and upper bounds, each residual capacity is nonnegative. Recall that the maximum flow algorithm described in this chapter (and the ones described in Chapter 7) works with only residual capacities and does not need arc flows, capacities, or lower bounds. Therefore we can use any of these algorithms to establish a maximum flow in the network. These algorithms terminate with optimal residual capacities. From these residual capacities we can construct maximum flow in a large number of ways. For example, through a change of variables we can reduce the computations to a situation we have considered before. For all arcs (i, j), let $u'_{ij} = u_{ij} - l_{ij}$, $r'_{ij} = r_{ij}$, and $x'_{ij} = x_{ij} - l_{ij}$. The residual capacity for arc (i, j) is $r_{ij} = (u_{ij} - x_{ij}) + (x_{ji} - l_{ji})$. Equivalently, $r'_{ij} = u'_{ij} - x'_{ij} + x'_{ji}$. Similarly, $r'_{ji} = u'_{ji} - x'_{ji} + x'_{ij}$. If we compute the x' values in terms of r' and u', we obtain the same expression as before, i.e., $x'_{ij} = \max(u'_{ij} - r'_{ij}, 0)$ and $x'_{ji} = \max(u'_{ji} -$

$r'_{ij}, 0)$. Converting back into the original variables, we obtain the following formulae:

$$x_{ij} = l_{ij} + \max(u_{ij} - r_{ij} - l_{ij}, 0),$$
$$x_{ji} = l_{ji} + \max(u_{ji} - r_{ji} - l_{ji}, 0).$$

We now show that the solution determined by this modified procedure solves the maximum flow problem with nonnegative lower bounds. Let x denote a feasible flow in G with value equal to v. Moreover, let $[S, \bar{S}]$ denote an s–t cut. We define the *capacity* of an s–t cut $[S, \bar{S}]$ as

$$u[S, \bar{S}] = \sum_{(i,j)\in(S,\bar{S})} u_{ij} - \sum_{(i,j)\in(\bar{S},S)} l_{ij}. \tag{6.7}$$

The capacity of the cut denotes the maximum amount of "net" flow that can be sent out of the node set S. We next use equality (6.3), which we restate for convenience.

$$v = \sum_{(i,j)\in(S,\bar{S})} x_{ij} - \sum_{(i,j)\in(\bar{S},S)} x_{ij}. \tag{6.8}$$

Substituting $x_{ij} \le u_{ij}$ in the first summation and $l_{ij} \le x_{ij}$ in the second summation of this inequality shows that

$$v \le \sum_{(i,j)\in(S,\bar{S})} u_{ij} - \sum_{(i,j)\in(\bar{S},S)} l_{ij} = u[S, \bar{S}]. \tag{6.9}$$

Inequality (6.9) indicates that the maximum flow value is less than or equal to the capacity of any s–t cut. At termination, the maximum flow algorithm obtains an s–t cut $[S, \bar{S}]$ with $r_{ij} = 0$ for every arc $(i, j) \in (S, \bar{S})$. Let x denote the corresponding flow with value equal to v. Since $r_{ij} = (u_{ij} - x_{ij}) + (x_{ji} - l_{ji})$, the conditions $x_{ij} \le u_{ij}$ and $l_{ji} \le x_{ji}$ imply that $x_{ij} = u_{ij}$ and $x_{ji} = l_{ji}$. Consequently, $x_{ij} = u_{ij}$ for every arc $(i, j) \in (S, \bar{S})$ and $x_{ij} = l_{ij}$ for every arc $(i, j) \in (\bar{S}, S)$. Substituting these values in (6.8), we find that

$$v = u[S, \bar{S}] = \sum_{(i,j)\in(S,\bar{S})} u_{ij} - \sum_{(i,j)\in(\bar{S},S)} l_{ij}. \tag{6.10}$$

In view of inequality (6.9), equation (6.10) implies that $[S, \bar{S}]$ is a minimum s–t cut and x is a maximum flow. As a by-product of this result, we have proved a generalization of the max-flow min-cut theorem for problems with nonnegative lower bounds.

Theorem 6.10 (Generalized Max-Flow Min-Cut Theorem). *If the capacity of an s–t cut $[S, \bar{S}]$ in a network with both lower and upper bounds on arc flows is defined by* (6.7), *the maximum value of flow from node s to node t equals the minimum capacity among all s–t cuts.* ◆

Establishing a Feasible Flow

We now address the issue of determining a feasible flow in the network. We first transform the maximum flow problem into a circulation problem by adding an arc (t, s) of infinite capacity. This arc carries the flow sent from node s to node t back to node s. Consequently, in the circulation formulation of the problem, the outflow of each node, including nodes s and t, equals its flow. Clearly, the maximum flow problem admits a feasible flow if and only if the circulation problem admits a feasible flow. Given the possibility of making this transformation, we now focus our intention on finding a feasible circulation, and characterizing conditions when an arbitrary circulation problem, with lower and upper bounds of flows, possesses a feasible solution.

The feasible circulation problem is to identify a flow x satisfying the following constraints:

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = 0 \qquad \text{for all } i \in N, \tag{6.11a}$$

$$l_{ij} \le x_{ij} \le u_{ij} \qquad \text{for all } (i,j) \in A. \tag{6.11b}$$

By replacing $x_{ij} = x'_{ij} + l_{ij}$ in constraints (6.11a) and (6.11b), we obtain the following transformed problem:

$$\sum_{\{j:(i,j)\in A\}} x'_{ij} - \sum_{\{j:(j,i)\in A\}} x'_{ji} = b(i) \qquad \text{for all } i \in N, \tag{6.12a}$$

$$0 \le x'_{ij} \le u_{ij} - l_{ij} \qquad \text{for all } (i,j) \in A, \tag{6.12b}$$

with supplies/demands $b(\cdot)$ at the nodes defined by

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

Observe that $\sum_{i\in N} b(i) = 0$ since each l_{ij} occurs twice in this expression, once with a positive sign and once with a negative sign. The feasible circulation problem is then equivalent to determining whether the transformed problem has a solution x' satisfying (6.12).

Notice that this problem is essentially the same as the feasible flow problem discussed in Application 6.1. In discussing this application we showed that by solving a maximum flow problem we either determine a solution satisfying (6.12) or show that no solution satisfies (6.12). If x'_{ij} is a feasible solution of (6.12), $x_{ij} = x'_{ij} + l_{ij}$ is a feasible solution of (6.11).

Characterizing a Feasible Flow

We next characterize feasible circulation problems (i.e., derive the necessary and sufficiency conditions for a circulation problem to possess a feasible solution). Let S be any set of nodes in the network. By summing the mass balance constraints of the nodes in S, we obtain the expression

$$\sum_{(i,j)\in(S,\bar{S})} x_{ij} - \sum_{(i,j)\in(\bar{S},S)} x_{ij} = 0. \tag{6.13}$$

Using the inequalities $x_{ij} \le u_{ij}$ in the first term of (6.13) and the inequalities $x_{ij} \ge l_{ij}$ in the second term, we find that

$$\sum_{(i,j)\in(\bar{S},S)} l_{ij} \le \sum_{(i,j)\in(S,\bar{S})} u_{ij}. \tag{6.14}$$

The expression in (6.14), which is a necessary condition for feasibility, states that the maximum amount of flow that we can send out from a set S of nodes must be at least as large as the minimum amount of flow that the nodes in S must receive. Clearly, if a set of nodes must receive more than what the other nodes can send them, the network has no feasible circulation. As we will see, these conditions are also sufficient for ensuring feasibility [i.e., if the network data satisfies the conditions (6.14) for every set S of nodes, the network has a feasible circulation that satisfies the flow bounds on all its arcs].

We give an algorithmic proof for the sufficiency of condition (6.14). The algorithm starts with a circulation x that satisfies the mass balance and capacity constraints, but might violate some of the lower bound constraints. The algorithm gradually converts this circulation into a feasible flow or identifies a node set S that violates condition (6.14).

With respect to a flow x, we refer to an arc (i, j) as *infeasible* if $x_{ij} < l_{ij}$ and *feasible* if $l_{ij} \leq x_{ij}$. The algorithm selects an infeasible arc (p, q) and attempts to make it feasible by increasing the flow on this arc. The mass balance constraints imply that in order to increase the flow on the arc, we must augment flow along one or more cycles in the residual network that contain arc (p, q) as a forward arc. We define the residual network $G(x)$ with respect to a flow x the same way we defined it previously except that we set the residual capacity of any infeasible arc (i, j) to the value $u_{ij} - x_{ij}$. Any augmenting cycle containing arc (p, q) as a forward arc must consist of a directed path in the residual network $G(x)$ from node q to node p plus the arc (p, q). We can use a labeling algorithm to identify a directed path from node q to node p.

We apply this procedure to one infeasible arc at a time, at each step decreasing the infeasibility of the arcs until we either identify a feasible flow or the labeling algorithm is unable to identify a directed path from node q to node p for some infeasible arc (p, q). We show that in the latter case, the maximum flow problem must be infeasible. Let S be the set of nodes labeled by the last application of the labeling algorithm. Clearly, $q \in S$ and $p \in \bar{S} \equiv N - S$. Since the labeling algorithm cannot label any node not in S, every arc (i, j) from S to $\bar{S}$ has a residual capacity of value zero. Therefore, $x_{ij} = u_{ij}$ for every arc $(i, j) \in (S, \bar{S})$ and $x_{ij} \leq l_{ij}$ for every arc $(i, j) \in (\bar{S}, S)$. Also observe that $(p, q) \in (\bar{S}, S)$ and $x_{pq} < l_{pq}$. Substituting these values in (6.13), we find that

$$\sum_{(i,j)\in(\bar{S},S)} l_{ij} > \sum_{(i,j)\in(S,\bar{S})} u_{ij},$$

contradicting condition (6.14), which we have already shown is necessary for feasibility. We have thus established the following fundamental theorem.

Theorem 6.11 (Circulation Feasibility Conditions). *A circulation problem with nonnegative lower bounds on arc flows is feasible if and only if for every set S of nodes*

$$\sum_{(i,j)\in(\bar{S},S)} l_{ij} \leq \sum_{(i,j)\in(S,\bar{S})} u_{ij}. \qquad \blacklozenge$$

Note that the proof of this theorem specifies a one pass algorithm, starting with the zero flow, for finding a feasible solution to any circulation problem whose arc upper bounds u_{ij} are all nonnegative. In Exercise 6.7 we ask the reader to specify a one pass algorithm for any situation (i.e., even when some upper bounds are negative).

A by-product of Theorem 6.11 is the following result, which states necessary and sufficiency conditions for the existence of a feasible solution for the feasible flow problem stated in (6.2). (Recall that a feasible flow problem is the feasibility version of the minimum cost flow problem.) We discuss the proof of this result in Exercise 6.43.

Theorem 6.12. *The feasible flow problem stated in (6.2) has a feasible solution if and only if for every subset* $S \subseteq N$, $b(S) - u[S, \bar{S}] \leq 0$, *where* $b(S) = \sum_{i \in S} b(i)$. ◆

6.8 SUMMARY

In this chapter we studied two closely related problems: the maximum flow problem and the minimum cut problem. After illustrating a variety of applications of these problems, we showed that the maximum flow and the minimum cut problems are closely related of each other (in fact, they are dual problems) and solving the maximum flow problem also solves the minimum cut problem. We began by showing that the value of any flow is less than or equal to the capacity of any cut in the network (i.e., this is a "weak duality" result). The fact that the value of some flow equals the capacity of some cut in the network (i.e., the "strong duality" result) is a deeper result. This result is known as the *max-flow min-cut theorem.* We establish it by specifying a labeling algorithm that maintains a feasible flow x in the network and sends additional flow along directed paths from the source node to the sink node in the residual network $G(x)$. Eventually, $G(x)$ contains no directed path from the source to the sink. At this point, the value of the flow x equals the capacity of some cut $[S, \bar{S}]$ in the network. The weak duality result implies that x is a maximum flow and $[S, \bar{S}]$ is a minimum cut. Since the labeling algorithm maintains an integer flow at every step (assuming integral capacity data), the optimal flow that it finds is integral. This result is a special case of a more general network flow integrality result that we establish in Chapter 9. The labeling algorithm runs in $O(nmU)$ time. This time bound is not attractive from the worst-case perspective. In Chapter 7 we develop two polynomial-time implementations of the labeling algorithm.

The max-flow min-cut theorem has far-reaching implications. It allows us to prove several important results in combinatorics that appear difficult to prove using other means. We proved the following results: (1) the maximum number of arc-disjoint (or node-disjoint) paths connecting two nodes s and t in a network equals the minimum number of arcs (or nodes) whose removal from the network leaves no directed path from node s to node t; and (2) in a bipartite network, the maximum cardinality of any matching equals the minimum cardinality of any node cover. In the exercises we ask the reader to prove other implications of the max-flow min-cut theorem.

To conclude this chapter we studied the maximum flow problem with nonnegative lower bounds on arc flows. We can solve this problem using a two-phase approach. The first phase determines a feasible flow if one exists, and the second phase converts this flow into a maximum flow; in both phases we solve a maximum flow problem with zero lower bounds. We also described a theoretical result for characterizing when a maximum flow problem with nonnegative lower bounds has a feasible solution. Roughly speaking, this characterization states that the maximum flow problem has a feasible solution if and only if the maximum possible outflow of every cut is at least as large as the minimum required inflow for that cut.

REFERENCE NOTES

The seminal paper of Ford and Fulkerson [1956a] on the maximum flow problem established the celebrated max-flow min-cut theorem. Fulkerson and Dantzig [1955], and Elias, Feinstein, and Shannon [1956] independently established this result. Ford and Fulkerson [1956a] and Elias et al. [1956] solved the maximum flow problem by augmenting path algorithms, whereas Fulkerson and Dantzig [1955] solved it by specializing the simplex method for linear programming. The labeling algorithm that we described in Section 6.5 is due to Ford and Fulkerson [1956a]; their classical book, Ford and Fulkerson [1962], offers an extensive treatment of this algorithm. Unfortunately, the labeling algorithm runs in pseudopolynomial time; moreover, as shown by Ford and Fulkerson [1956a], for networks with arbitrary irrational arc capacities, the algorithm can perform an infinite sequence of augmentations and might converge to a value different from the maximum flow value. Several improved versions of the labeling algorithm overcome this limitation. We provide citations to these algorithms and to their improvements in the reference notes of Chapter 7. In Chapter 7 we also discuss computational properties of maximum flow algorithms.

In Section 6.6 we studied the combinatorial implications of the max-flow min-cut theorem. Theorems 6.7 and 6.8 are known as Menger's theorem. Theorem 6.9 is known as the König-Egerváry theorem. Ford and Fulkerson [1962] discuss these and several additional combinatorial results that are provable using the max-flow min-cut theorem.

In Section 6.7 we studied the feasibility of a network flow problem with nonnegative lower bounds imposed on the arc flows. Theorem 6.11 is due to Hoffman [1960], and Theorem 6.12 is due to Gale [1957]. The book by Ford and Fulkerson [1962] discusses these and some additional feasibility results extensively. The algorithm we have presented for identifying a feasible flow in a network with nonnegative lower bounds is adapted from this book.

The applications of the maximum flow problem that we described in Section 6.2 are adapted from the following papers:

1. Feasible flow problem (Berge and Ghouila-Houri [1962])
2. Problem of representatives (Hall [1956])
3. Matrix rounding problem (Bacharach [1966])
4. Scheduling on uniform parallel machines (Federgruen and Groenevelt [1986])
5. Distributed computing on a two-processor model (Stone [1977])
6. Tanker scheduling problem (Dantzig and Fulkerson [1954])

Elsewhere in this book we describe other applications of the maximum flow problem. These applications include: (1) the tournament problem (Application 1.3, Ford and Johnson [1959]), (2) the police patrol problem (Exercise 1.9, Khan [1979]),

(3) nurse staff scheduling (Exercise 6.2, Khan and Lewis [1987]), (4) solving a system of equations (Exercise 6.4, Lin [1986]), (5) statistical security of data (Exercises 6.5, Application 8.3, Gusfield [1988], Kelly, Golden, and Assad [1990]), (6) the minimax transportation problem (Exercise 6.6, Ahuja [1986]), (7) the baseball elimination problem (Application 8.1, Schwartz [1966]), (8) network reliability testing (Application 8.2, Van Slyke and Frank [1972]), (9) open pit mining (Application 19.1, Johnson [1968]), (10) selecting freight handling terminals (Application 19.2, Rhys [1970]), (11) optimal destruction of military targets (Application 19.3, Orlin [1987]), (12) the flyaway kit problem (Application 19.4, Mamer and Smith [1982]), (13) maximum dynamic flows (Application 19.12, Ford and Fulkerson [1958a]), and (14) models for building evacuation (Application 19.13, Chalmet, Francis, and Saunders [1982]).

Two other interesting applications of the maximum flow problem are preemptive scheduling on machines with different speeds (Martel [1982]), and the multifacility rectilinear distance location problem (Picard and Ratliff [1978]). The following papers describe additional applications or provide additional references: McGinnis and Nuttle [1978], Picard and Queyranne [1982], Abdallaoui [1987], Gusfield, Martel, and Fernandez-Baca [1987], Gusfield and Martel [1989], and Gallo, Grigoriadis, and Tarjan [1989].

EXERCISES

6.1. **Dining problem.** Several families go out to dinner together. To increase their social interaction, they would like to sit at tables so that no two members of the same family are at the same table. Show how to formulate finding a seating arrangement that meets this objective as a maximum flow problem. Assume that the dinner contingent has p families and that the ith family has $a(i)$ members. Also assume that q tables are available and that the jth table has a seating capacity of $b(j)$.

6.2. **Nurse staff scheduling** (Khan and Lewis [1987]). To provide adequate medical service to its constituents at a reasonable cost, hospital administrators must constantly seek ways to hold staff levels as low as possible while maintaining sufficient staffing to provide satisfactory levels of health care. An urban hospital has three departments: the emergency room (department 1), the neonatal intensive care nursery (department 2), and the orthopedics (department 3). The hospital has three work shifts, each with different levels of necessary staffing for nurses. The hospital would like to identify the minimum number of nurses required to meet the following three constraints: (1) the hospital must allocate at least 13, 32, and 22 nurses to the three departments (over all shifts); (2) the hospital must assign at least 26, 24, and 19 nurses to the three shifts (over all departments); and (3) the minimum and maximum number of nurses allocated to each department in a specific shift must satisfy the following limits:

		Department		
		1	2	3
	1	(6, 8)	(11, 12)	(7, 12)
Shift	2	(4, 6)	(11, 12)	(7, 12)
	3	(2, 4)	(10, 12)	(5, 7)

Suggest a method using maximum flows to identify the minimum number of nurses required to satisfy all the constraints.

6.3. A commander is located at one node p in a communication network G and his subordinates are located at nodes denoted by the set S. Let u_{ij} be the effort required to eliminate arc (i, j) from the network. The problem is to determine the minimal effort required to block all communications between the commander and his subordinates. How can you solve this problem in polynomial time?

6.4. **Solving a system of equations** (Lin [1986]). Let $F = \{f_{ij}\}$ be a given $p \times q$ matrix and consider the following system of $p + q$ equations in the (possibly fractional) variables y:

$$\sum_{j=1}^{q} f_{ij}y_{ij} = u_i, \qquad 1 \le i \le p, \tag{6.15a}$$

$$\sum_{i=1}^{p} f_{ij}y_{ij} = v_j, \qquad 1 \le j \le q. \tag{6.15b}$$

In this system $u_i \ge 0$ and $v_j \ge 0$ are given constants satisfying the condition $\sum_{i=1}^{p} u_i = \sum_{j=1}^{q} v_j$.

(a) Define a matrix $D = \{d_{ij}\}$ as follows: $d_{ij} = 0$ if $f_{ij} = 0$, and $d_{ij} = 1$ if $f_{ij} \neq 0$. Show that (6.15) has a feasible solution if and only if the following system of $p + q$ equations has a feasible solution x:

$$\sum_{j=1}^{q} d_{ij}x_{ij} = u_i, \qquad 1 \le i \le p, \tag{6.16a}$$

$$\sum_{i=1}^{p} d_{ij}x_{ij} = v_j, \qquad 1 \le j \le q. \tag{6.16b}$$

(b) Show how to formulate the problem of identifying a feasible solution of the system (6.16) as a feasible circulation problem (i.e., identifying a circulation in some network with lower and upper bounds imposed on the arc flows). [*Hint*: The network has a node i for the ith row in (6.16a), a node $\bar{j}$ for the jth row in (6.16b), and one extra node s.]

6.5. **Statistical security of data** (Kelly, Golden, and Assad [1990], and Gusfield [1988]). The U.S. Census Bureau produces a variety of tables from its census data. Suppose that it wishes to produce a $p \times q$ table $D = \{d_{ij}\}$ of nonnegative integers. Let $r(i)$ denote the sum of the matrix elements in the ith row and let $c(j)$ denote the sum of the matrix elements in the jth column. Assume that each sum $r(i)$ and $c(j)$ is strictly positive. The Census Bureau often wishes to disclose all the row and column sums along with some matrix elements (denoted by a set Y) and yet suppress the remaining elements to ensure the confidentiality of privileged information. Unless it exercises care, by disclosing the elements in Y, the Bureau might permit someone to deduce the exact value of one or more of the suppressed elements. It is possible to deduce a suppressed element d_{ij} if only one value of d_{ij} is consistent with the row and column sums and the disclosed elements in Y. We say that any such suppressed element is *unprotected*. Describe a polynomial-time algorithm for identifying all the unprotected elements of the matrix and their values.

6.6. **Minimax transportation problem** (Ahuja [1986]). Suppose that $G = (N, A)$ is an uncapacitated transportation problem (as defined in Section 1.2) and that we want to find an integer flow x that minimizes the objective function $\max\{c_{ij}x_{ij}: (i, j) \in A\}$ among all feasible integer flows.

(a) Consider a relaxed version of the minimax transportation problem: Given a parameter λ, we want to know whether some feasible flow satisfies the condition $\max\{c_{ij}x_{ij}: (i, j) \in A\} \le \lambda$. Show how to solve this problem as a maximum flow problem. [*Hint*: Use the condition $\max\{c_{ij}x_{ij} : (i, j) \in A\} \leq \lambda$ to formulate the problem as a feasible flow problem.]

(b) Use the result in part (a) to develop a polynomial-time algorithm for solving the minimax transportation problem. What is the running time of your algorithm?

6.7. Consider a generalization of the feasible flow problem discussed in Application 6.1. Suppose that the flow bounds constraints are $l_{ij} \leq x_{ij} \leq u_{ij}$ instead of $0 \leq x_{ij} \leq u_{ij}$ for some nonnegative l_{ij}. How would you solve this generalization of the feasible flow problem as a single maximum flow problem?

6.8 Consider a generalization of the problem that we discussed in Application 6.2. Suppose that each club must nominate one of its members as a town representative so that the number of representatives belonging to the political party P_k is between l_k and u_k. Formulate this problem as a maximum flow problem with nonnegative lower bounds on arc flows.

6.9. In the example concerning the scheduling of uniform parallel machines (Application 6.4), we assumed that the same number of machines are available each day. How would you model a situation when the number of available machines varies from day to day? Illustrate your method on the example given in Application 6.4. Assume that three machines are available on days 1, 2, 4, and 5; two machines on days 3 and 6; and four machines on the rest of the days.

6.10. Can you solve the police patrol problem described in Exercise 1.9 using a maximum flow algorithm. If so, how?

6.11. Suppose that we wish to partition an undirected graph into two components with the minimum number of arcs between the components. How would you solve this problem?

6.12. Consider the network shown in Figure 6.22(a) together with the feasible flow x given in the figure.

(a) Specify four s–t cuts in the network, each containing four forward arcs. List the capacity, residual capacity, and the flow across each cut.

(b) Draw the residual network for the network given in Figure 6.22(a) and list four augmenting paths from node s to node t.

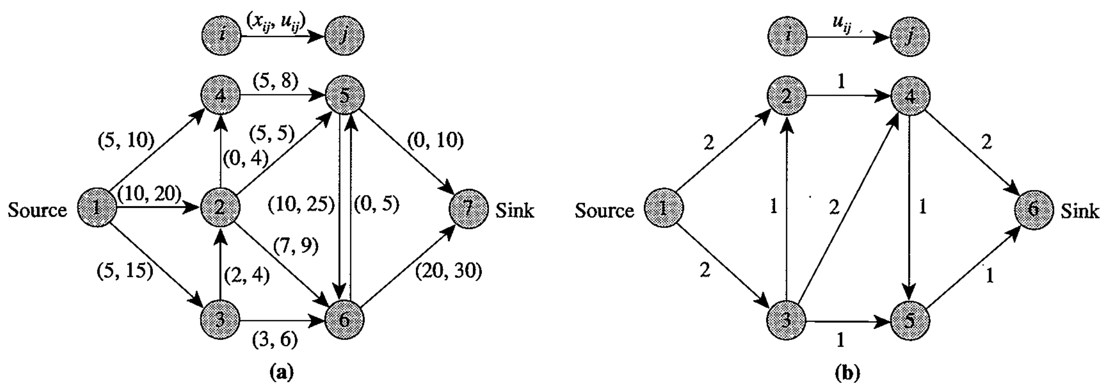


Figure 6.22 Examples for Exercises 6.12 and 6.13.

6.13. Solve the maximum flow problem shown in Figure 6.22(b) by the labeling algorithm, augmenting flow along the longest path in the residual network (i.e., the path containing maximum number of arcs). Specify the residual network before each augmentation. After every augmentation, decompose the flow into flows along directed paths from node s to node t. Finally, specify the minimum cut in the network obtained by the labeling algorithm.

6.14. Use the labeling algorithm to establish a maximum flow in the undirected network shown in Figure 6.23. Show the residual network at the end of each augmentation and specify the minimum cut that the algorithm obtains when it terminates.

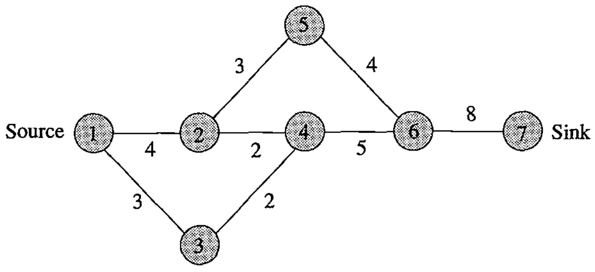


Figure 6.23 Example for Exercise 6.14.

6.15. Consider the network given in Figure 6.24; assume that each arc has capacity 1.

(a) Compute the maximum number of arc-disjoint paths from the source node to the sink node. (You might do so by inspection.)

(b) Enumerate all s–t cuts in the network. For each s–t cut $[S, \bar{S}]$, list the node partition and the sets of forward and backward arcs.

(c) Verify that the maximum number of arc-disjoint paths from node s to node t equals the minimum number of forward arcs in an s–t cut.

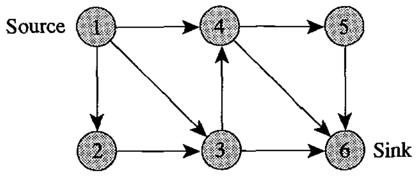


Figure 6.24 Example for Exercise 6.15.

6.16. Consider the matrix rounding problem given below (see Application 6.3). We want to round each element in the matrix, and also the row and column sums, to the nearest multiple of 2 so that the sum of the rounded elements in each row equals the rounded row sum and the sum of the rounded elements in each column equals the rounded column sum. Formulate this problem as a maximum flow problem and solve it.

				Row sum
	7.5	6.3	15.4	29.2
	3.9	9.1	3.6	16.6
	15.0	5.5	21.5	42.0
Column sum	26.4	20.9	40.5	

6.17. Formulate the following example of the scheduling problem on uniform parallel machines that we discussed in Application 6.4 as a maximum flow problem. Solve the problem by the labeling algorithm, assuming that two machines are available each day.

Job (j)	1	2	3	4
Processing time (p_j) (in days)	2.5	3.1	5.0	1.8
Release time (r_j)	1	5	0	2
Due date (d_j)	3	7	6	5

6.18. Minimum flow problem. The *minimum flow problem* is a close relative of the maximum flow problem with nonnegative lower bounds on arc flows. In the minimum flow problem, we wish to send the minimum amount of flow from the source to the sink, while satisfying given lower and upper bounds on arc flows.

(a) Show how to solve the minimum flow problem by using two applications of any maximum flow algorithm that applies to problems with zero lower bounds on arc flows. (*Hint*: First construct a feasible flow and then convert it into a minimum flow.)

(b) Prove the following *min-flow max-cut theorem*. Let the *floor* (or lower bound on the cut capacity) of an s–t cut $[S, \bar{S}]$ be defined as $\sum_{(i,j)\in(S,\bar{S})} l_{ij} - \sum_{(i,j)\in(\bar{S},S)} u_{ij}$. Show that the minimum value of all the flows from node s to node t equals the maximum floor of all s–t cuts.

6.19. Machine setup problem. A job shop needs to perform eight tasks on a particular day. Figure 6.25(a) shows the start and end times of each task. The workers must perform

Task	Start time	End time
1	1:00 P.M.	1:30 P.M.
2	6:00 P.M.	8:00 P.M.
3	10:00 P.M.	11:00 P.M.
4	4:00 P.M.	5:00 P.M.
5	4:00 P.M.	7:00 P.M.
6	12:00 noon	1:00 P.M.
7	2:00 P.M.	5:00 P.M.
8	11:00 P.M.	12:00 midnight

(a)

	1	2	3	4	5	6	7	8
1	—	60	10	25	30	20	15	40
2	10	—	40	55	40	5	30	35
3	65	30	—	0	45	30	20	5
4	0	50	35	—	20	15	10	20
5	20	24	40	50	—	15	5	23
6	10	8	9	35	12	—	30	30
7	15	30	6	18	15	30	—	10
8	20	35	15	12	75	13	25	—

(b)

Figure 6.25 Machine setup data: (a) task start and end times; (b) setup times in transforming between tasks.

these tasks according to this schedule so that exactly one worker performs each task. A worker cannot work on two jobs at the same time. Figure 6.25(b) shows the setup time (in minutes) required for a worker to go from one task to another. We wish to find the minimum number of workers to perform the tasks. Formulate this problem as a minimum flow problem (see Exercise 6.18).

6.20. Show how to transform a maximum flow problem having several source nodes and several sink nodes to one with only one source node and one sink node.

6.21. Show that if we add any number of incoming arcs, with any capacities, to the source node, the maximum flow value remains unchanged. Similarly, show that if we add any number of outgoing arcs, with any capacities, at the sink node, the maximum flow value remains unchanged.

6.22. Show that the maximum flow problem with integral data has a finite optimal solution if and only if the network contains no infinite capacity directed path from the source node to the sink node.

6.23. Suppose that a network has some infinite capacity arcs but no infinite capacity paths from the source to the sink. Let A^0 denote the set of arcs with finite capacities. Show that we can replace the capacity of each infinite capacity arc by a finite number $M \geq \sum_{(i,j)\in A^0} u_{ij}$ without affecting the maximum flow value.

6.24. Suppose that you want to solve a maximum flow problem containing parallel arcs, but the maximum flow code you own cannot handle parallel arcs. How would you use the code to solve your maximum flow problem?

6.25. **Networks with node capacities.** In some networks, in addition to arc capacities, each node i, other than the source and the sink, might have an upper bound, say $w(i)$, on the flow that can pass through it. For example, the nodes might be airports with limited runway capacity for takeoff and landings, or might be switches in a communication network with a limited number of ports. In these networks we are interested in determining the maximum flow satisfying both the arc and node capacities. Transform this problem to the standard maximum flow problem. From the perspective of worst-case complexity, is the maximum flow problem with upper bounds on nodes more difficult to solve than the standard maximum flow problem?

6.26. Suppose that a maximum flow network contains a node, other than the source node, with no incoming arc. Can we delete this node without affecting the maximum flow value? Similarly, can we delete a node, other than the sink node, with no outgoing arc?

6.27. Suppose that you are asked to solve a maximum flow problem in a directed network subject to the absolute value flow bound constraints $-u_{ij} \leq x_{ij} \leq u_{ij}$ imposed on some arcs (i, j). How would you solve this problem?

6.28. Suppose that a maximum flow is available. Show how you would find a minimum cut in $O(m)$ additional time. Suppose, instead, that a minimum cut is available. Could you use this cut to obtain a maximum flow faster than applying a maximum flow algorithm?

6.29. **Painted network theorem.** Let G be a directed network with a distinguished arc (s, t). Suppose that we paint each arc in the network as green, yellow, or red, with arc (s, t) painted yellow. Show that the painted network satisfies exactly one of the following two cases: (1) arc (s, t) is contained in a cycle of yellow and green arcs in which all yellow arcs have the same direction but green arcs can have arbitrary directions; (2) arc (s, t) is contained in a cut of yellow and red arcs in which all yellow arcs have the same direction but red arcs can have arbitrary directions.

6.30. Show that if $x_{ij} = u_{ij}$ for some arc (i, j) in every maximum flow, this arc must be a forward arc in some minimum cut.

6.31. An engineering department consisting of p faculty members, $F_1, F_2, \ldots, F_p$, will offer p courses, $C_1, C_2, \ldots, C_p$, in the coming semester and each faculty member will teach exactly one course. Each faculty member ranks two courses he (or she) would like to teach, ranking them according to his (or her) preference.

(a) We say that a course assignment is a *feasible* assignment if every faculty member teaches a course within his (or her) preference list. How would you determine whether the department can find a feasible assignment? (For a related problem see Exercise 12.46.)

(b) A feasible assignment is said to be *k-feasible* if it assigns at most k faculty members to their second most preferred courses. For a given k, suggest an algorithm for determining a k-feasible assignment.

(c) We say that a feasible assignment is an *optimal assignment* if it maximizes the number of faculty members assigned to their most preferred course. Suggest an algorithm for determining an optimal assignment and analyze its complexity. [*Hint*: Use the algorithm in part (b) as a subroutine.]

6.32. **Airline scheduling problem.** An airline has p flight legs that it wishes to service by the fewest possible planes. To do so, it must determine the most efficient way to combine these legs into flight schedules. The starting time for flight i is a_i and the finishing time is b_i. The plane requires r_{ij} hours to return from the point of destination of flight i to the point of origin of flight j. Suggest a method for solving this problem.

6.33. A flow x is *even* if for every arc $(i, j) \in A$, x_{ij} is an even number; it is *odd* if for every $(i, j) \in A$, x_{ij} is an odd number. Either prove that each of the following claims are true or give a counterexample for them.

(a) If all arc capacities are even, the network has an even maximum flow.

(b) If all arc capacities are odd, the network has an odd maximum flow.

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

(a) If x_{ij} is a maximum flow, either $x_{ij} = 0$ or $x_{ji} = 0$ for every arc $(i, j) \in A$.

(b) Any network always has a maximum flow x for which, for every arc $(i, j) \in A$, either $x_{ij} = 0$ or $x_{ji} = 0$.

(c) If all arcs in a network have different capacities, the network has a unique minimum cut.

(d) In a directed network, if we replace each directed arc by an undirected arc, the maximum flow value remains unchanged.

(e) If we multiply each arc capacity by a positive number λ, the minimum cut remains unchanged.

(f) If we add a positive number λ to each arc capacity, the minimum cut remains unchanged.

6.35. **(a)** Suppose that after solving a maximum flow problem you realize that you have underestimated the capacity of an arc (p, q) by k units. Show that the labeling algorithm can reoptimize the problem in $O(km)$ time.

(b) Suppose that instead of underestimating the capacity of the arc (p, q), you had overestimated its capacity by k units. Can you reoptimize the problem in $O(km)$ time?

6.36. **(a)** Construct a family of networks with the number of s–t cuts growing exponentially with n.

(b) Construct a family of networks with the number of minimum cuts growing exponentially with n.

6.37. **(a)** Given a maximum flow in a network, describe an algorithm for determining the minimum cut $[S, \bar{S}]$ with the property that for every other minimum cut $[R, \bar{R}]$, $R \subseteq S$.

(b) Describe an algorithm for determining the minimum cut $[S, \bar{S}]$ with the property that for every other minimum cut $[R, \bar{R}]$, $S \subseteq R$.

(c) Describe an algorithm for determining whether the maximum flow problem has a unique minimum cut.

6.38. Let $[S, \bar{S}]$ and $[T, \bar{T}]$ be two s–t cuts in the directed network G. Show that the cut capacity function $u[.\,,.]$ is *submodular*, that is, $u[S, \bar{S}] + u[T, \bar{T}] \geq u[S \cup T, \overline{S \cup T}] + u[S \cap T, \overline{S \cap T}]$. (*Hint*: Prove this result by case analysis.)

6.39. Show that if $[S, \bar{S}]$ and $[T, \bar{T}]$ are both minimum cuts, so are $[S \cup T, \overline{S \cup T}]$ and $[S \cap T, \overline{S \cap T}]$.

6.40. Suppose that we know a noninteger maximum flow in a directed network with integer arc capacities. Suggest an algorithm for converting this flow into an integer maximum flow. What is the running time of your algorithm? (*Hint*: Send flows along cycles.)

6.41. **Optimal coverage of sporting events.** A group of reporters want to cover a set of sporting events in an olympiad. The sports events are held in several stadiums throughout a city. We known the starting time of each event, its duration, and the stadium where it is held. We are also given the travel times between different stadiums. We want to determine the least number of reporters required to cover the sporting events. How would you solve this problem?

6.42. In Section 6.7 we showed how to solve the maximum flow problem in directed networks with nonnegative lower bounds by solving two maximum flow problems with zero lower flow bounds. Try to generalize this approach for undirected networks in which the flow on any arc (i, j) is permitted in either direction, but whichever direction is chosen the amount of flow is at least l_{ij}. If you succeed in developing an algorithm, state the algorithm along with a proof that it correctly solves the problem; if you do not succeed in developing an algorithm state reasons why the generalization does not work.

6.43. **Feasibility of the feasible flow problem** (Gale [1957]). Show that the feasible flow problem, discussed in Application 6.1, has a feasible solution if and only if for every subset $S \subseteq N$, $b(S) - u[S, \bar{S}] \leq 0$. (*Hint*: Transform the feasible flow problem into a circulation problem with nonzero lower bounds and use the result of Theorem 6.11.)

6.44. Prove Theorems 6.7 and 6.8 for undirected networks.

6.45. Let N^+ and N^- be two nonempty disjoint node sets in G. Describe a method for determining the maximum number of arc-disjoint paths from N^+ to N^- (i.e., each path can start at any node in N^+ and can end at any node in N^-). What is the implication of the max-flow min-cut theorem in this case? (*Hint*: Generalize the statement of Theorem 6.7.)

6.46. Consider a 0–1 matrix **H** with n_1 rows and n_2 columns. We refer to a row or a column of the matrix **H** as a line. We say that a set of 1's in the matrix **H** is *independent* if no two of them appear in the same line. We also say that a set of lines in the matrix is a *cover* of **H** if they include (i.e., "cover") all the 1's in the matrix. Show that the maximum number of independent 1's equals the minimum number of lines in a cover. (*Hint*: Use the max-flow min-cut theorem on an appropriately defined network.)

6.47. In a directed acyclic network G, certain arcs are colored blue, while others are colored red. Consider the problem of covering the blue arcs by directed paths, which can start and end at any node (these paths can contain arcs of any color). Show that the minimum number of directed paths needed to cover the blue arcs is equal to the maximum number of blue arcs that satisfy the property that no two of these arcs belong to the same path. Will this result be valid if G contains directed cycles? (*Hint*: Use the min-flow max-cut theorem stated in Exercise 6.18.)

6.48. **Pathological example for the labeling algorithm.** In the residual network $G(x)$ corresponding to a flow x, we define an *augmenting walk* as a directed walk from node s to node t that visits any arc at most once (it might visit nodes multiple times—in particular, an augmenting walk might visit nodes s and t multiple times.)

(a) Consider the network shown in Figure 6.26(a) with the arcs labeled a, b, c and d; note that one arc capacity is irrational. Show that this network contains an infinite sequence of augmenting walks whose residual capacities sum to the maximum flow value. (*Hint*: Each augmenting walk of the sequence contains exactly two arcs from node s to node t with finite residual capacities.)

(b) Now consider the network shown in Figure 6.26(b). Show that this network contains an infinite sequence of augmenting walks whose residual capacities sum to a value different than the maximum flow value.

(c) Next consider the network shown in Figure 6.26(c); in addition to the arcs shown, the network contain an infinite capacity arc connecting each node pair in the set

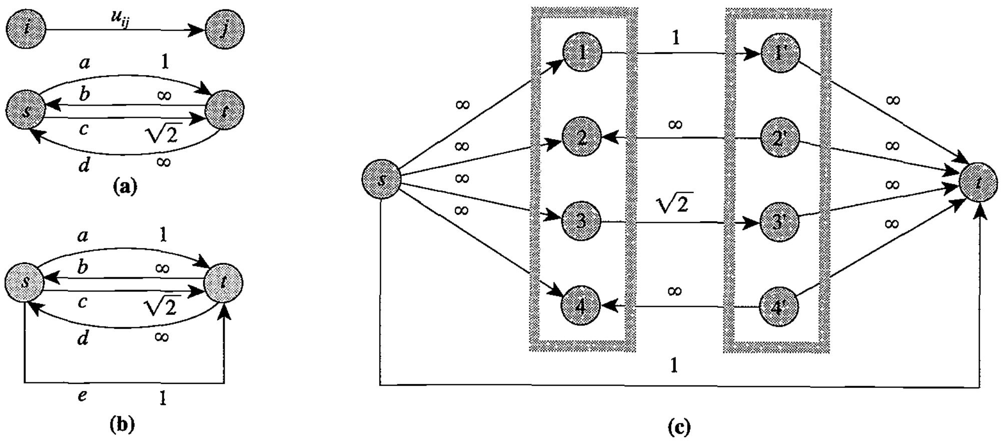


Figure 6.26 A subgraph of a pathological instance for labeling algorithm. The fall graph contains an infinite capacity are connecting each pair of nodes i and j as well as each pair of nodes i′ and j′.

$\{1, 2, 3, 4\}$ and each node pair in the set $\{1', 2', 3', 4'\}$. Show that each augmenting walk in the solution of part (b) corresponds to an augmenting path in Figure 6.26(c). Conclude that the labeling algorithm, when applied to a maximum flow problem with irrational capacities, might perform an infinite sequence of augmentations and the terminal flow value might be different than the maximum flow value.

7

MAXIMUM FLOWS: POLYNOMIAL ALGORITHMS

Every day, in every way, I am getting better and better.
—Émile Coué

Chapter Outline

7.1 INTRODUCTION

The generic augmenting path algorithm that we discussed in Chapter 6 is a powerful tool for solving maximum flow problems. Not only is it guaranteed to solve any maximum flow problem with integral capacity data, it also provides us with a constructive tool for establishing the fundamental max-flow min-cut theorem and therefore for developing many useful combinatorial applications of network flow theory.

As we noted in Chapter 6, however, the generic augmenting path algorithm has two significant computational limitations: (1) its worst-case computational complexity of $O(nmU)$ is quite unattractive for problems with large capacities; and (2) from a theoretical perspective, for problems with irrational capacity data, the algorithm might converge to a nonoptimal solution. These limitations suggest that the algorithm is not entirely satisfactory, in theory. Unfortunately, the algorithm is not very satisfactory in practice as well: On very large problems, it can require too much solution time.

Motivated by a desire to develop methods with improved worst-case complexity and empirical behavior, in this chapter we study several refinements of the generic augmenting path algorithm. We also introduce and study another class of algorithms, known as *preflow-push algorithms*, that have recently emerged as the most powerful techniques, both theoretically and computationally, for solving maximum flow problems.

Before describing these algorithms and analyzing them in detail, let us pause to reflect briefly on the theoretical limits of maximum flow algorithms and to introduce the general solution strategies employed by the refined augmenting path algorithms that we consider in this chapter. Flow decomposition theory shows that, in principle, we might be able to design augmenting path algorithms that are capable of finding a maximum flow in no more than m augmentations. For suppose that x is an optimal flow and x° is any initial flow (possibly the zero flow). By the flow decomposition property (see Section 3.5), we can obtain x from x° by a sequence of (1) at most m augmentations on augmenting paths from node s to node t, plus (2) flows around augmenting cycles. If we define x' as the flow vector obtained from x° by sending flows along only the augmenting paths, x' is also a maximum flow (because flows around augmenting cycles do not change the flow value into the sink node). This observation demonstrates a theoretical possibility of finding a maximum flow using at most m augmentations. Unfortunately, to apply this flow decomposition argument, we need to know a maximum flow. As a consequence, no algorithm developed in the literature achieves this theoretical bound of m augmentations. Nevertheless, it is possible to improve considerably on the $O(nU)$ bound on the number of augmentations required by the generic augmenting path algorithm.

How might we attempt to reduce the number of augmentations or even eliminate them altogether? In this chapter we consider three basic approaches:

1. Augmenting in "large" increments of flow
2. Using a combinatorial strategy that limits the type of augmenting paths we can use at each step
3. Relaxing the mass balance constraint at intermediate steps of the algorithm, and thus not requiring that each flow change must be an augmentation that starts at the source node and terminates at the sink node

Let us now consider each of these approaches. As we have seen in Chapter 6, the generic augmenting path algorithm could be slow because it might perform a large number of augmentations, each carrying a small amount of flow. This observation suggests one natural strategy for improving the augmenting path algorithm: Augment flow along a path with a *large* residual capacity so that the number of augmentations remains relatively *small*. The *maximum capacity augmenting path algorithm* uses this idea: It always augments flow along a path with the maximum residual capacity. In Section 7.3 we show that this algorithm performs $O(m \log U)$ augmentations. A variation of this algorithm that augments flows along a path with a *sufficiently large*, but not necessarily maximum residual capacity also performs $O(m \log U)$ augmentations and is easier to implement. We call this algorithm the *capacity scaling algorithm* and describe it in Section 7.3.

Another possible strategy for implementing and improving the augmenting path algorithm would be to develop an approach whose implementation is entirely independent of the arc capacity data and relies on a combinatorial argument for its convergence. One such approach would be somehow to restrict the choice of augmenting paths in some way. In one such approach we might always augment flow along a "shortest path" from the source to the sink, defining a shortest path as a directed path in the residual network consisting of the fewest number of arcs. If we augment flow along a shortest path, the length of any shortest path either stays the same or increases. Moreover, within m augmentations, the length of the shortest path is guaranteed to increase. (We prove these assertions in Section 7.4.) Since no path contains more than $n - 1$ arcs, this result guarantees that the number of augmentations is at most $(n - 1)m$. We call this algorithm the *shortest augmenting path algorithm* and discuss it in Section 7.4.

The preflow-push algorithms use the third strategy we have identified: They seek out "shortest paths" as in the shortest augmenting path algorithm, but do not send flow along paths from the source to the sink. Instead, they send flows on individual arcs. This "localized" strategy, together with clever rules for implementing this strategy, permits these algorithms to obtain a speed-up not obtained by any augmenting path algorithm. We study these preflow-push algorithms in Sections 7.6 through 7.9.

The concept of distance labels is an important construct used to implement the shortest augmenting path algorithm and the preflow-push algorithms that we consider in this chapter. So before describing the improved algorithms, we begin by discussing this topic.

7.2 DISTANCE LABELS

A *distance function* d: $N \rightarrow Z^{+} \cup \{0\}$ with respect to the residual capacities r_{ij} is a function from the set of nodes to the set of nonnegative integers. We say that a distance function is *valid* with respect to a flow x if it satisfies the following two conditions:

$$d(t) = 0; \tag{7.1}$$

$$d(i) \le d(j) + 1 \quad \text{for every arc } (i, j) \text{ in the residual network } G(x). \tag{7.2}$$

We refer to $d(i)$ as the *distance label* of node i and conditions (7.1) and (7.2) as the *validity conditions*. The following properties show why the distance labels might be of use in designing network flow algorithms.

Property 7.1. *If the distance labels are valid, the distance label $d(i)$ is a lower bound on the length of the shortest (directed) path from node i to node t in the residual network.*

To establish the validity of this observation, let $i = i_1 - i_2 - \cdots - i_k - i_{k+1} = t$ be any path of length k from node i to node t in the residual network. The validity conditions imply that

$$\begin{aligned} d(i_k) &\le d(i_{k+1}) + 1 = d(t) + 1 = 1, \\ d(i_{k-1}) &\le d(i_k) + 1 \le 2, \\ d(i_{k-2}) &\le d(i_{k-1}) + 1 \le 3, \\ &\vdots \\ d(i) = d(i_1) &\le d(i_2) + 1 \le k. \end{aligned}$$

Property 7.2. *If $d(s) \geq n$, the residual network contains no directed path from the source node to the sink node.*

The correctness of this observation follows from the facts that $d(s)$ is a lower bound on the length of the shortest path from s to t in the residual network, and therefore no directed path can contain more than $(n - 1)$ arcs. Therefore, if $d(s) \geq n$, the residual network contains no directed path from node s to node t.

We now introduce some additional notation. We say that the distance labels are *exact* if for each node i, $d(i)$ equals the length of the shortest path from node i to node t in the residual network. For example, in Figure 7.1, if node 1 is the source node and node 4 is the sink node, then $d = (0, 0, 0, 0)$ is a valid vector of distance label, and $d = (3, 1, 2, 0)$ is a vector of exact distance labels. We can determine exact distance labels for all nodes in $O(m)$ time by performing a backward breadth-first search of the network starting at the sink node (see Section 3.4).

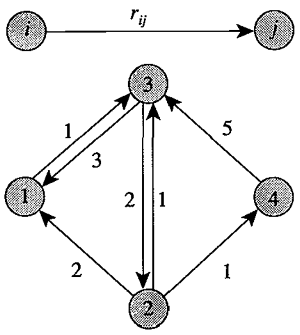


Figure 7.1 Residual network.

Admissible Arcs and Admissible Paths

We say that an arc (i, j) in the residual network is *admissible* if it satisfies the condition that $d(i) = d(j) + 1$; we refer to all other arcs as *inadmissible*. We also refer to a path from node s to node t consisting entirely of admissible arcs as an *admissible path*. Later, we use the following property of admissible paths.

Property 7.3. *An admissible path is a shortest augmenting path from the source to the sink.*

Since every arc (i,j) in an admissible path P is admissible, the residual capacity of this arc and the distance labels of its end nodes satisfy the conditions (1) $r_{ij} > 0$, and (2) $d(i) = d(j) + 1$. Condition (1) implies that P is an augmenting path and condition (2) implies that if P contains k arcs, then $d(s) = k$. Since $d(s)$ is a lower bound on the length of any path from the source to the sink in the residual network (from Property 7.1), the path P must be a shortest augmenting path.

7.3 CAPACITY SCALING ALGORITHM

We begin by describing the maximum capacity augmenting path algorithm and noting its computational complexity. This algorithm always augments flow along a path with the maximum residual capacity. Let x be any flow and let v be its flow value.

As before, let v^* be the maximum flow value. The flow decomposition property (i.e., Theorem 3.5), as applied to the residual network $G(x)$, implies that we can find m or fewer directed paths from the source to the sink whose residual capacities sum to $(v^* - v)$. Thus the maximum capacity augmenting path has residual capacity at least $(v^* - v)/m$. Now consider a sequence of $2m$ consecutive maximum capacity augmentations starting with the flow x. If each of these augmentations augments at least $(v^* - v)/2m$ units of flow, then within $2m$ or fewer iterations we will establish a maximum flow. Note, however, that if one of these $2m$ consecutive augmentations carries less than $(v^* - v)/2m$ units of flow, then from the initial flow vector x, we have reduced the residual capacity of the maximum capacity augmenting path by a factor of at least 2. This argument shows that within $2m$ consecutive iterations, the algorithm either establishes a maximum flow or reduces the residual capacity of the maximum capacity augmenting path by a factor of at least 2. Since the residual capacity of any augmenting path is at most $2U$ and is at least 1, after $O(m \log U)$ iterations, the flow must be maximum. (Note that we are essentially repeating the argument used to establish the geometric improvement approach discussed in Section 3.3.)

As we have seen, the maximum capacity augmentation algorithm reduces the number of augmentations in the generic labeling algorithm from $O(nU)$ to $O(m \log U)$. However, the algorithm performs more computations per iteration, since it needs to identify an augmenting path with the maximum residual capacity, not just any augmenting path. We now suggest a variation of the maximum capacity augmentation algorithm that does not perform more computations per iteration and yet establishes a maximum flow within $O(m \log U)$. Since this algorithm scales the arc capacities implicitly, we refer to it as the *capacity scaling algorithm.*

The essential idea underlying the capacity scaling algorithm is conceptually quite simple: We augment flow along a path with a *sufficiently large* residual capacity, instead of a path with the maximum augmenting capacity because we can obtain a path with a sufficiently large residual capacity fairly easily—in $O(m)$ time. To define the capacity scaling algorithm, let us introduce a parameter Δ and, with respect to a given flow x, define the *Δ-residual network* as a network containing arcs whose residual capacity is at least Δ. Let $G(x, \Delta)$ denote the Δ-residual network. Note that $G(x, 1) = G(x)$ and $G(x, \Delta)$ is a subgraph of $G(x)$. Figure 7.2 illustrates this definition. Figure 7.2(a) gives the residual network $G(x)$ and Figure 7.2(b) gives the Δ-residual network $G(x, \Delta)$ for $\Delta = 8$. Figure 7.3 specifies the capacity scaling algorithm.

Let us refer to a phase of the algorithm during which Δ remains constant as a *scaling phase* and a scaling phase with a specific value of Δ as a *Δ-scaling phase.* Observe that in a Δ-scaling phase, each augmentation carries at least Δ units of flow. The algorithm starts with $\Delta = 2^{\lfloor \log U \rfloor}$ and halves its value in every scaling phase until $\Delta = 1$. Consequently, the algorithm performs $1 + \lfloor \log U \rfloor = O(\log U)$ scaling phases. In the last scaling phase, $\Delta = 1$, so $G(x, \Delta) = G(x)$. This result shows that the algorithm terminates with a maximum flow.

The efficiency of the algorithm depends on the fact that it performs at most $2m$ augmentations per scaling phase. To establish this result, consider the flow at the end of the Δ-scaling phase. Let x' be this flow and let v' denote its flow value. Furthermore, let S be the set of nodes reachable from node s in $G(x', \Delta)$. Since

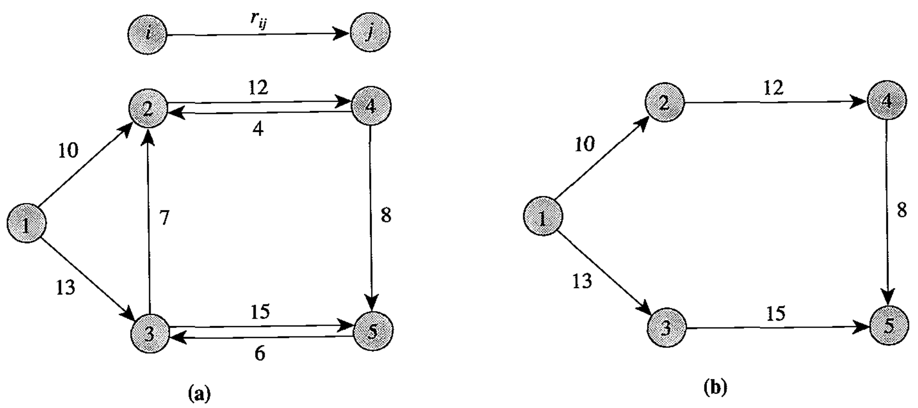


Figure 7.2 Illustrating the Δ-residual network: (a) residual network $G(x)$; (b) Δ-residual network $G(x, \Delta)$ for $\Delta = 8$.

```
algorithm capacity scaling;
begin
    x : = 0;
    Δ : = 2^⌊log U⌋;
    while Δ ≥ 1 do
    begin
        while G(x, Δ) contains a path from node s to node t do
        begin
            identify a path P in G(x, Δ);
            δ : = min{r_ij : (i, j) ∈ P};
            augment δ units of flow along P and update G(x, Δ);
        end;
        Δ : = Δ/2;
    end;
end;
```

Figure 7.3 Capacity scaling algorithm.

$G(x', \Delta)$ contains no augmenting path from the source to the sink, $t \notin S$. Therefore, $[S, \bar{S}]$ forms an s–t cut. The definition of S implies that the residual capacity of every arc in $[S, \bar{S}]$ is strictly less than Δ, so the residual capacity of the cut $[S, \bar{S}]$ is at most $m\Delta$. Consequently, $v^* - v' \le m\Delta$ (from Property 6.2). In the next scaling phase, each augmentation carries at least $\Delta/2$ units of flow, so this scaling phase can perform at most $2m$ such augmentations. The labeling algorithm described in Section 6.5 requires $O(m)$ time to identify an augmenting path, and updating the Δ-residual network also requires $O(m)$ time. These arguments establish the following result.

Theorem 7.4. *The capacity scaling algorithm solves the maximum flow problem within $O(m \log U)$ augmentations and runs in $O(m^2 \log U)$ time.* ◆

It is possible to reduce the complexity of the capacity scaling algorithm even further—to $O(nm \log U)$—using ideas of the shortest augmenting path algorithm, described in the next section.

7.4 SHORTEST AUGMENTING PATH ALGORITHM

The shortest augmenting path algorithm always augments flow along a shortest path from the source to the sink in the residual network. A natural approach for implementing this approach would be to look for shortest paths by performing a breadth first search in the residual network. If the labeling algorithm maintains the set L of labeled nodes as a queue, then by examining the labeled nodes in a first-in, first-out order, it would obtain a shortest path in the residual network (see Exercise 3.30). Each of these iterations would require $O(m)$ steps in the worst case, and (by our subsequent observations) the resulting computation time would be $O(nm^2)$. Unfortunately, this computation time is excessive. We can improve it by exploiting the fact that the minimum distance from any node i to the sink node t is monotonically nondecreasing over all augmentations. By fully exploiting this property, we can reduce the average time per augmentation to $O(n)$.

The shortest augmenting path algorithm proceeds by augmenting flows along admissible paths. It constructs an admissible path incrementally by adding one arc at a time. The algorithm maintains a *partial admissible path* (i.e., a path from s to some node i consisting solely of admissible arcs) and iteratively performs *advance* or *retreat* operations from the last node (i.e., the tip) of the partial admissible path, which we refer to as the *current node*. If the current node i has (i.e., is incident to) an admissible arc (i, j), we perform an advance operation and add arc (i, j) to the partial admissible path; otherwise, we perform a retreat operation and backtrack one arc. We repeat these operations until the partial admissible path reaches the sink node at which time we perform an augmentation. We repeat this process until the flow is maximum. Before presenting a formal description of the algorithm, we illustrate it on the numerical example given in Figure 7.4(a).

We first compute the initial distance labels by performing the backward breadth-first search of the residual network starting at the sink node. The numbers next to the nodes in Figure 7.4(a) specify these values of the distance labels. In this example we adopt the convention of selecting the arc (i, j) with the smallest value of j whenever node i has several admissible arcs. We start at the source node with a null partial admissible path. The source node has several admissible arcs, so we perform an advance operation. This operation adds the arc (1, 2) to the partial admissible path. We store this path using predecessor indices, so we set pred(2) = 1. Now node 2 is the current node and the algorithm performs an advance operation at node 2. In doing so, it adds arc (2, 7) to the partial admissible path, which now becomes 1–2–7. We also set pred(7) = 2. In the next iteration, the algorithm adds arc (7, 12) to the partial admissible path obtaining 1–2–7–12, which is an admissible path to the sink node. We perform an augmentation of value $\min\{r_{12}, r_{27}, r_{7,12}\} = \min\{2, 1, 2\} = 1$, and thus saturate the arc (2, 7). Figure 7.4(b) specifies the residual network at this stage.

We again start at the source node with a null partial admissible path. The algorithm adds the arc (1, 2) and node 2 becomes the new current node. Now we find that node 2 has no admissible arc. To create new admissible arcs, we must increase the distance label of node 2. We thus increase $d(2)$ to the value $\min\{d(j) + 1 : (i, j) \in A(i) \text{ and } r_{ij} > 0\} = \min\{d(1) + 1\} = 4$. We refer to this operation as a *relabel* operation. We will later show that a relabel operation preserves the validity

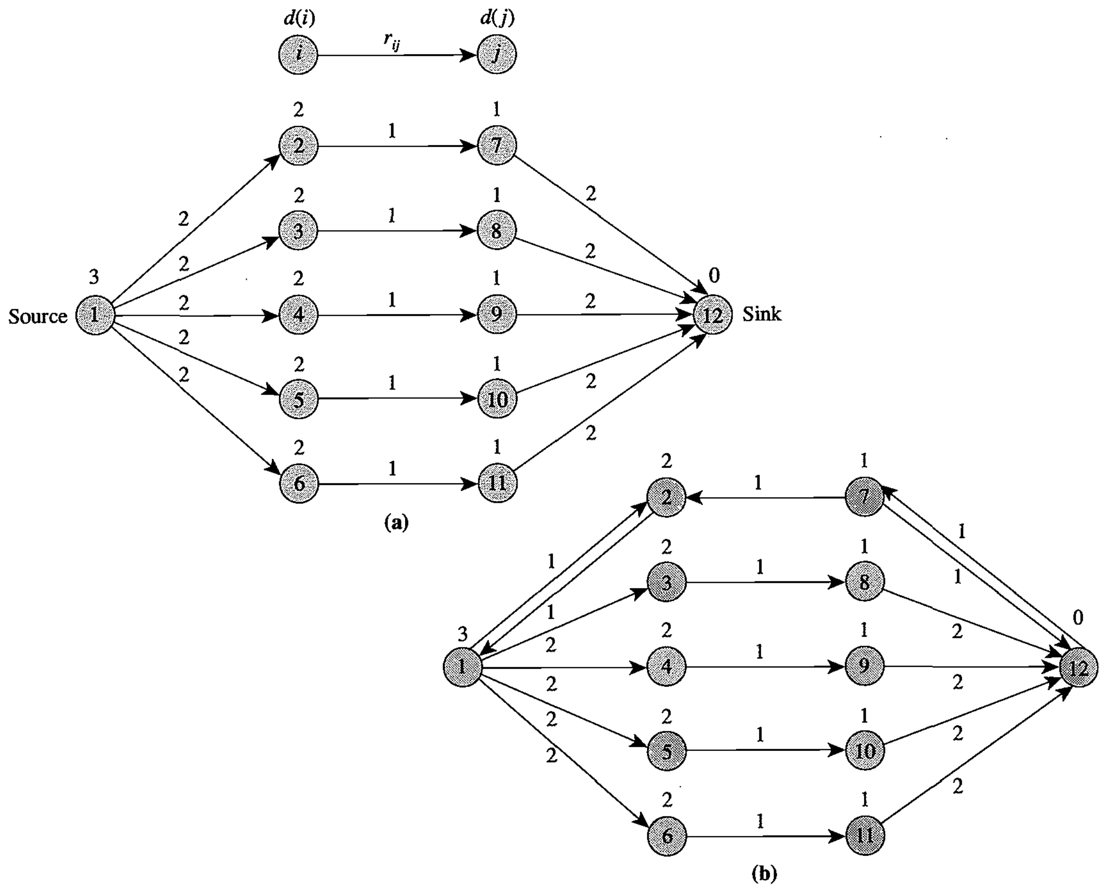


Figure 7.4 Illustrating the shortest augmenting path algorithm.

conditions imposed upon the distance labels. Observe that the increase in $d(2)$ causes arc (1, 2) to become inadmissible. Thus we delete arc (1, 2) from the partial admissible path which again becomes a null path. In the subsequent operations, the algorithm identifies the admissible paths 1–3–8–12, 1–4–9–12, 1–5–10–12, and 1–6–11–12 and augments unit flows on these paths. We encourage the reader to carry out the details of these operations. Figures 7.5 and 7.6 specify the details of the algorithm.

Correctness of the Algorithm

In our analysis of the shortest augmenting path algorithm we first show that it correctly solves the maximum flow problem.

Lemma 7.5. *The shortest augmenting path algorithm maintains valid distance labels at each step. Moreover, each relabel (or, retreat) operation strictly increases the distance label of a node.*

```
algorithm shortest augmenting path;
begin
    $x := 0$;
    obtain the exact distance labels $d(i)$;
    $i := s$;
    while $d(s) < n$ do
    begin
        if $i$ has an admissible arc then
            begin
                advance(i);
                if $i = t$ then augment and set $i = s$
            end
        else retreat(i)
    end;
end;
```

Figure 7.5 Shortest augmenting path algorithm.

```
procedure advance(i);
begin
    let $(i, j)$ be an admissible arc in $A(i)$;
    pred$(j) := i$ and $i := j$;
end;
```

(a)

```
procedure retreat(i);
begin
    $d(i) := \min\{d(j) + 1 : (i, j) \in A(i) \text{ and } r_{ij} > 0\}$;
    if $i \neq s$ then $i :=$ pred$(i)$;
end;
```

(b)

```
procedure augment;
begin
    using the predecessor indices identify an augmenting
    path $P$ from the source to the sink;
    $\delta := \min\{r_{ij} : (i, j) \in P\}$;
    augment $\delta$ units of flow along path $P$;
end;
```

(c)

Figure 7.6 Procedures of the shortest augmenting path algorithm.

Proof. We show that the algorithm maintains valid distance labels at every step by performing induction on the number of augment and relabel operations. (The advance operation does not affect the admissibility of any arc because it does not change any residual capacity or distance label.) Initially, the algorithm constructs valid distance labels. Assume, inductively, that the distance labels are valid prior to an operation (i.e., they satisfy the validity conditions). We need to check whether these conditions remain valid (a) after an augment operation, and (b) after a relabel operation.

(a) Although a flow augmentation on arc (i, j) might remove this arc from the residual network, this modification to the residual network does not affect the validity of the distance labels for this arc. An augmentation on arc (i, j) might, however, create an additional arc (j, i) with $r_{ji} > 0$ and therefore also create an additional inequality $d(j) \leq d(i) + 1$ that the distance labels must satisfy. The distance labels satisfy this validity condition, though, since $d(i) = d(j) + 1$ by the admissibility property of the augmenting path.

(b) The relabel operation modifies $d(i)$; therefore, we must show that each incoming and outgoing arc at node i satisfies the validity conditions with respect to the new distance labels, say $d'(i)$. The algorithm performs a relabel operation at node i when it has no admissible arc; that is, no arc $(i, j) \in A(i)$ satisfies the conditions $d(i) = d(j) + 1$ and $r_{ij} > 0$. This observation, in light of the validity condition $d(i) \leq d(j) + 1$, implies that $d(i) < d(j) + 1$ for all arcs $(i, j) \in A$ with a positive residual capacity. Therefore, $d(i) < \min\{d(j) + 1 : (i, j) \in A(i) \text{ and } r_{ij} > 0\} = d'(i)$, which is the new distance label after the relabel operation. We have thus shown that relabeling preserves the validity condition for all arcs emanating from node i, and that each relabel operation strictly increases the value of $d(i)$. Finally, note that every incoming arc (k, i) satisfies the inequality $d(k) \leq d(i) + 1$ (by the induction hypothesis). Since $d(i) < d'(i)$, the relabel operation again preserves validity condition for arc (k, i). ◆

The shortest augmenting path algorithm terminates when $d(s) \geq n$, indicating that the network contains no augmenting path from the source to the sink (from Property 7.2). Consequently, the flow obtained at the end of the algorithm is a maximum flow. We have thus proved the following theorem.

Theorem 7.6. *The shortest augmenting path algorithm correctly computes a maximum flow.* ◆

Complexity of the Algorithm

We now show that the shortest augmenting path algorithm runs in $O(n^2m)$ time. We first describe a data structure used to select an admissible arc emanating from a given node. We call this data structure the *current-arc data structure*. Recall that we used this data structure in Section 3.4 in our discussion of search algorithms. We also use this data structure in almost all the maximum flow algorithms that we describe in subsequent sections. Therefore, we review this data structure before proceeding.

Recall that we maintain the arc list $A(i)$ which contains all the arcs emanating from node i. We can arrange the arcs in these lists arbitrarily, but the order, once decided, remains unchanged throughout the algorithm. Each node i has a *current arc*, which is an arc in $A(i)$ and is the next candidate for admissibility testing. Initially, the current arc of node i is the first arc in $A(i)$. Whenever the algorithm attempts to find an admissible arc emanating from node i, it tests whether the node's current arc is admissible. If not, it designates the next arc in the arc list as the current arc. The algorithm repeats this process until either it finds an admissible arc or reaches the end of the arc list.

Consider, for example, the arc list of node 1 in Figure 7.7. In this instance, $A(1) = \{(1, 2), (1, 3), (1, 4), (1, 5), (1, 6)\}$. Initially, the current arc of node 1 is arc (1, 2). Suppose that the algorithm attempts to find an admissible arc emanating from node 1. It checks whether the node's current arc, arc (1, 2), is admissible. Since it is not, the algorithm designates arc (1, 3) as the current arc of node 1. The arc (1, 3) is also inadmissible, so the current arc becomes arc (1, 4), which is admissible. From this point on, arc (1, 4) remains the current arc of node 1 until it becomes inadmissible because the algorithm has increased the value of $d(4)$ or decreased the value of the residual capacity of arc (1, 4) to zero.

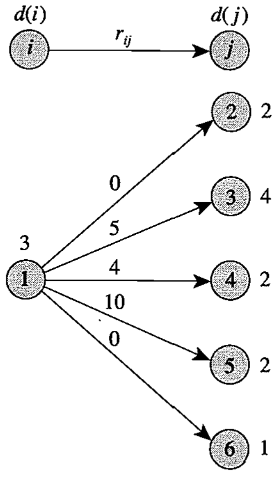


Figure 7.7 Selecting admissible arcs emanating from a node.

Let us consider the situation when the algorithm reaches the end of the arc list without finding any admissible arc. Can we say that $A(i)$ has no admissible arc? We can, because it is possible to show that if an arc (i, j) is inadmissible in previous iterations, it remains inadmissible until $d(i)$ increases (see Exercise 7.13). So if we reach the end of the arc list, we perform a relabel operation and again set the current arc of node i to be the first arc in $A(i)$. The relabel operation also examines each arc in $A(i)$ once to compute the new distance label, which is same as the time it spends in identifying admissible arcs at node i in one scan of the arc list. We have thus established the following result.

***Property** 7.7. If the algorithm relabels any node at most k times, the total time spent in finding admissible arcs and relabeling the nodes is* $O(k \sum_{i \in N} | A(i) |) = O(km)$.

We shall be using this result several times in this and the following chapters. We also use the following result in several places.

***Lemma 7.8.** If the algorithm relabels any node at most k times, the algorithm saturates arcs (i.e., reduces their residual capacity to zero) at most km/2 times.*

Proof. We show that between two consecutive saturations of an arc (i, j), both $d(i)$ and $d(j)$ must increase by at least 2 units. Since, by our hypothesis, the algorithm increases each distance label at most k times, this result would imply that the algorithm could saturate any arc at most $k/2$ times. Therefore, the total number of arc saturations would be $km/2$, which is the assertion of the lemma.

Suppose that an augmentation saturates an arc (i, j). Since the arc (i, j) is admissible,

$$d(i) = d(j) + 1. \tag{7.3}$$

Before the algorithm saturates this arc again, it must send flow back from node j to node i. At this time, the distance labels $d'(i)$ and $d'(j)$ satisfy the equality

$$d'(j) = d'(i) + 1. \tag{7.4}$$

In the next saturation of arc (i, j), we must have

$$d''(i) = d''(j) + 1. \tag{7.5}$$

Using (7.3) and (7.4) in (7.5), we see that

$$d''(i) = d''(j) + 1 \geq d'(j) + 1 = d'(i) + 2 \geq d(i) + 2.$$

The inequalities in this expression follow from Lemma 7.5. Similarly, it is possible to show that $d''(j) \geq d(j) + 2$. As a result, between two consecutive saturations of the arc (i, j), both $d(i)$ and $d(j)$ increase by at least 2 units, which is the conclusion of the lemma. ◆

Lemma 7.9.

(a) *In the shortest augmenting path algorithm each distance label increases at most n times. Consequently, the total number of relabel operations is at most n^2.*

(b) *The number of augment operations is at most $nm/2$.*

Proof. Each relabel operation at node i increases the value of $d(i)$ by at least 1 unit. After the algorithm has relabeled node i at most n times, $d(i) \geq n$. From this point on, the algorithm never again selects node i during an advance operation since for every node k in the partial admissible path, $d(k) < d(s) < n$. Thus the algorithm relabels a node at most n times and the total number of relabel operations is bounded by n^2. In view of Lemma 7.8, the preceding result implies that the algorithm saturates at most $nm/2$ arcs. Since each augmentation saturates at least one arc, we immediately obtain a bound of $nm/2$ on the number of augmentations. ◆

Theorem 7.10. *The shortest augmenting path algorithm runs in $O(n^2m)$* time.

Proof. Using Lemmas 7.9 and 7.7 we find that the total effort spent in finding admissible arcs and in relabeling the nodes is $O(nm)$. Lemma 7.9 implies that the total number of augmentations is $O(nm)$. Since each augmentation requires $O(n)$ time, the total effort for the augmentation operations is $O(n^2m)$. Each retreat operation relabels a node, so the total number of retreat operations is $O(n^2)$. Each advance operation adds one arc to the partial admissible path, and each retreat operation deletes one arc from it. Since each partial admissible path has length at most n, the algorithm requires at most $O(n^2 + n^2m)$ advance operations. The first term comes from the number of retreat (relabel) operations, and the second term from the number of augmentations. The combination of these bounds establishes the theorem. ◆

A Practical Improvement

The shortest augmenting path algorithm terminates when $d(s) \geq n$. This termination criteria is satisfactory for the worst-case analysis but might not be efficient in practice. Empirical investigations have revealed that the algorithm spends too much time relabeling nodes and that a major portion of this effort is performed after the algorithm has established a maximum flow. This happens because the algorithm does not know that it has found a maximum flow. We next suggest a technique that is capable of detecting the presence of a minimum cut and so the existence of a maximum flow much before the label of node s satisfies the condition $d(s) \geq n$. Incorporating this technique in the shortest augmenting path algorithm improves its performance substantially in practice.

We illustrate this technique by applying it to the numerical example we used earlier to illustrate the shortest augmenting path algorithm. Figure 7.8 gives the residual network immediately after the last augmentation. Although the flow is now a maximum flow, since the source is not connected to the sink in the residual network, the termination criteria of $d(1) \geq 12$ is far from being satisfied. The reader can verify that after the last augmentation, the algorithm would increase the distance labels of nodes 6, 1, 2, 3, 4, 5, in the given order, each time by 2 units. Eventually, $d(1) \geq 12$ and the algorithm terminates. Observe that the node set S of the minimum cut $[S, \bar{S}]$ equals {6, 1, 2, 3, 4, 5}, and the algorithm increases the distance labels of all the nodes in S without performing any augmentation. The technique we describe essentially detects a situation like this one.

To implement this approach, we maintain an n-dimensional additional array,

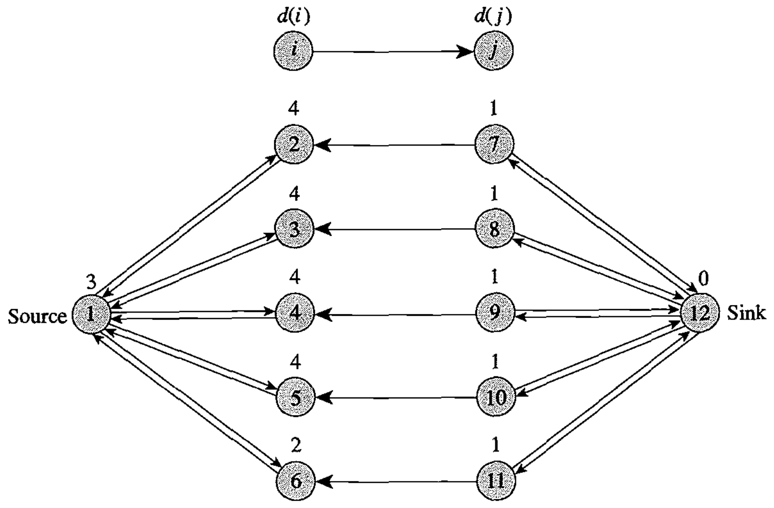


Figure 7.8 Bad example for the shortest augmenting path algorithm.

numb, whose indices vary from 0 to $(n - 1)$. The value numb(k) is the number of nodes whose distance label equals k. The algorithm initializes this array while computing the initial distance labels using a breadth first search. At this point, the positive entries in the array numb are consecutive [i.e., the entries numb(0), numb(1), . . . , numb(l) will be positive up to some index l and the remaining entries will all be zero]. For example, the numb array for the distance labels shown in Figure 7.8 is numb(0) = 1, numb(1) = 5, numb(2) = 1, numb(3) = 1, numb(4) = 4 and the remaining entries are zero. Subsequently, whenever the algorithm increases the distance label of a node from k_1 to k_2, it subtracts 1 from numb(k_1), adds 1 to numb(k_2) and checks whether numb$(k_1) = 0$. If numb(k_1) does equal zero, the algorithm terminates. As seen earlier, the shortest augmenting path algorithm augments unit flow along the paths 1–2–7–12, 1–3–8–12, 1–4–9–12, 1–5–10–12, and 1–6–11–12. At the end of these augmentations, we obtain the residual network shown in Figure 7.8. When we continue the shortest augmenting path algorithm from this point, it constructs the partial admissible path 1–6. Next it relabels node 6 and its distance label increases from 2 to 4. The algorithm finds that numb(2) = 0 and it terminates.

To see why this termination criterion works, let $S = \{i \in N: d(i) > k_1\}$ and $\bar{S} = \{i \in N: d(i) < k_1\}$. It is easy to verify that $s \in S$ and $t \in \bar{S}$. Now consider the s–t cut $[S, \bar{S}]$. The definitions of the sets S and $\bar{S}$ imply that $d(i) > d(j) + 1$ for all $(i, j) \in [S, \bar{S}]$. The validity condition (7.2) implies that $r_{ij} = 0$ for each arc $(i, j) \in [S, \bar{S}]$. Therefore, $[S, \bar{S}]$ is a minimum cut and the current flow is a maximum flow.

Application to Capacity Scaling Algorithm

In the preceding section we described an $O(m^2 \log U)$ time capacity scaling algorithm for the maximum flow problem. We can improve the running time of this algorithm to $O(nm \log U)$ by using the shortest augmenting path as a subroutine in the capacity scaling algorithm. Recall that the capacity scaling algorithm performs a number of Δ-scaling phases and in the Δ-scaling phase sends the maximum possible flow in the Δ-residual network $G(x, \Delta)$, using the labeling algorithm as a subroutine. In the improved implementation, we use the shortest augmenting path algorithm to send the maximum possible flow from node s to node t. We accomplish this by defining the distance labels with respect to the network $G(x, \Delta)$ and augmenting flow along the shortest augmenting path in $G(x, \Delta)$. Recall from the preceding section that a scaling phase contains $O(m)$ augmentations. The complexity analysis of the shortest augmenting path algorithm implies that if the algorithm is guaranteed to perform $O(m)$ augmentations, it would run in $O(nm)$ time because the time for augmentations reduces from $O(n^2m)$ to $O(nm)$ and all other operations, as before, require $O(nm)$ time. These observations immediately yield a bound of $O(nm \log U)$ on the running time of the capacity scaling algorithm.

Further Worst-Case Improvements

The idea of augmenting flows along shortest paths is intuitively appealing and easy to implement in practice. The resulting algorithms identify at most $O(nm)$ augmenting paths and this bound is tight [i.e., on particular examples these algorithms perform $\Omega(nm)$ augmentations]. The only way to improve the running time of the shortest augmenting path algorithm is to perform fewer computations per augmentation. The use of a sophisticated data structure, called *dynamic trees*, reduces the average time for each augmentation from $O(n)$ to $O(\log n)$. This implementation of the shortest augmenting path algorithm runs in $O(nm \log n)$ time, and obtaining further improvements appears quite difficult except in very dense networks. We describe the dynamic tree implementation of the shortest augmenting path algorithm in Section 8.5.

7.5 DISTANCE LABELS AND LAYERED NETWORKS

Like the shortest augmenting path algorithm, several other maximum flow algorithms send flow along shortest paths from the source to the sink. Dinic's algorithm is a popular algorithm in this class. This algorithm constructs shortest path networks, called *layered networks*, and establishes *blocking flows* (to be defined later) in these networks. In this section we point out the relationship between layered networks and distance labels. By developing a modification of the shortest augmenting path algorithm that reduces to Dinic's algorithm, we show how to use distance labels to simulate layered networks.

With respect to a given flow x, we define the *layered network* V as follows. We determine the exact distance labels d in $G(x)$. The layered network consists of those arcs (i, j) in $G(x)$ satisfying the condition $d(i) = d(j) + 1$. For example, consider the residual network $G(x)$ given in Figure 7.9(a). The number beside each node represents its exact distance label. Figure 7.9(b) shows the layered network of $G(x)$. Observe that by definition every path from the source to the sink in the layered network V is a shortest path in $G(x)$. Observe further that some arc in V might not be contained in any path from the source to the sink. For example, in Figure 7.9(b), arcs (5, 7) and (6, 7) do not lie on any path in V from the source to the sink. Since these arcs do not participate in any flow augmentation, we typically delete them from the layered network; doing so gives us Figure 7.9(c). In the resulting layered network, the nodes are partitioned into layers of nodes V_0, V_1, V_2, . . . , V_l; layer k contains the nodes whose distance labels equal k. Furthermore, for every

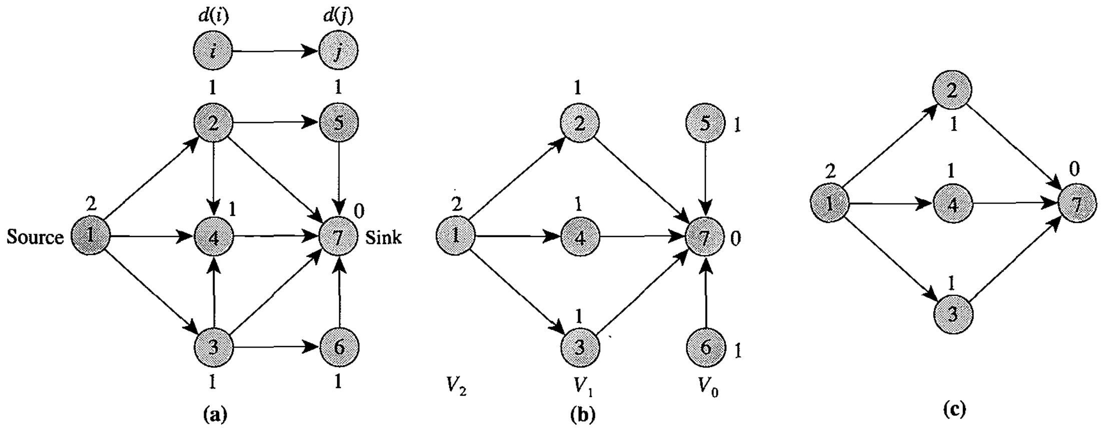


Figure 7.9 Forming layered networks: (a) residual network; (b) corresponding layered network; (c) layered network after deleting redundant arcs.

arc (i, j) in the layered network, $i \in V_k$ and $j \in V_{k-1}$ for some k. Let the source node have the distance label l.

Dinic's algorithm proceeds by augmenting flows along directed paths from the source to the sink in the layered network. The augmentation of flow along an arc (i, j) reduces the residual capacity of arc (i, j) and increases the residual capacity of the reversal arc (j, i); however, each arc of the layered network is admissible, and therefore Dinic's algorithm does not add reversal arcs to the layered network. Consequently, the length of every augmenting path is $d(s)$ and in an augmenting path every arc (i, j) has $i \in V_k$ and $j \in V_{k-1}$ for some k. The latter fact allows us to determine an augmenting path in the layered network, on average, in $O(n)$ time. (The argument used to establish the $O(n)$ time bound is the same as that used in our analysis of the shortest augmenting path algorithm.) Each augmentation saturates at least one arc in the layered network, and after at most m augmentations the layered network contains no augmenting path. We call the flow at this stage a *blocking flow*. We have shown that we can establish a blocking flow in a layered network in $O(nm)$ time.

When a blocking flow x has been established in a network, Dinic's algorithm recomputes the exact distance labels, forms a new layered network, and repeats these computations. The algorithm terminates when as it is forming the new layered networks, it finds that the source is not connected to the sink. It is possible to show that every time Dinic's algorithm forms a new layered network, the distance label of the source node strictly increases. Consequently, Dinic's algorithm forms at most n layered networks and runs in $O(n^2m)$ time.

We now show how to view Dinic's algorithm as a somewhat modified version of the shortest augmenting path algorithm. We make the following three modifications to the shortest augmenting path algorithm.

Modification 1. In operation retreat(i), we do not change the distance label of node i, but subsequently term node i as *blocked*. A blocked node has no admissible path to the sink node.

Modification 2. We define an arc (i, j) to be admissible if $d(i) = d(j) + 1$, $r_{ij} > 0$, and node j is not blocked.

Modification 3. When the source node is blocked, by performing a backward breadth-first search we recompute the distance labels of all nodes exactly.

We term the computations within two successive recomputations of distance labels as occurring within a single *phase*. We note the following facts about the modified shortest augmenting path algorithm.

1. At the beginning of a phase, when the algorithm recomputes the distance labels $d(\cdot)$, the set of admissible arcs defines a layered network.
2. Each arc (i, j) in the admissible path satisfies $d(i) = d(j) + 1$; therefore, arc (i, j) joins two successive layers of the layered network. As a result, every admissible path is an augmenting path in the layered network.
3. Since we do not update distance labels within a phase, every admissible path has length equal to $d(s)$.

4. The algorithm performs at most m augmentations within a phase because each augmentation causes at least one arc to become inadmissible by reducing its residual capacity to zero, and the algorithm does not create new admissible arcs.
5. A phase ends when the network contains no admissible path from node s to node t. Hence, when the algorithm recomputes distance labels at the beginning of the next phase, $d(s)$ must increase (why?).

The preceding facts show that the modified shortest augmenting path algorithm essentially reduces to Dinic's algorithm. They also show that the distance labels are sufficiently powerful to simulate layered networks. Further, they are simpler to understand than layered networks, easier to manipulate, and lead to more efficient algorithms in practice. Distance labels are also attractive because they are generic solution approaches that find applications in several different algorithms; for example, the generic preflow-push algorithm described next uses distance labels, as does many of its variants described later.

7.6 GENERIC PREFLOW-PUSH ALGORITHM

We now study a class of algorithms, known as *preflow-push* algorithms, for solving the maximum flow problem. These algorithms are more general, more powerful, and more flexible than augmenting path algorithms. The best preflow-push algorithms currently outperform the best augmenting path algorithms in theory as well as in practice. In this section we study the generic preflow-push algorithm. In the following sections we describe special implementations of the generic approach with improved worst-case complexity.

The inherent drawback of the augmenting path algorithms is the computationally expensive operation of sending flow along a path, which requires $O(n)$ time in the worst case. Preflow-push algorithms do not suffer from this drawback and obtain dramatic improvements in the worst-case complexity. To understand this point better, consider the (artificially extreme) example shown in Figure 7.10. When applied to this problem, any augmenting path algorithm would discover 10 augmenting paths, each of length 10, and would augment 1 unit of flow along each of these paths. Observe, however, that although all of these paths share the same first eight arcs, each augmentation traverses all of these arcs. If we could have sent 10 units of flow from node 1 to node 9, and then sent 1 unit of flow along 10 different paths of length 2, we would have saved the repetitive computations in traversing the common set of arcs. This is the essential idea underlying the preflow-push algorithms.

Augmenting path algorithms send flow by augmenting along a path. This basic operation further decomposes into the more elementary operation of sending flow along individual arcs. Thus sending a flow of δ units along a path of k arcs decomposes into k basic operations of sending a flow of δ units along each of the arcs of the path. We shall refer to each of these basic operations as a *push*. The preflow-push algorithms push flows on individual arcs instead of augmenting paths.

Because the preflow-push algorithms push flows along the individual arcs, these algorithms do not satisfy the mass balance constraints (6.1b) at intermediate stages. In fact, these algorithms permit the flow entering a node to exceed the flow leaving

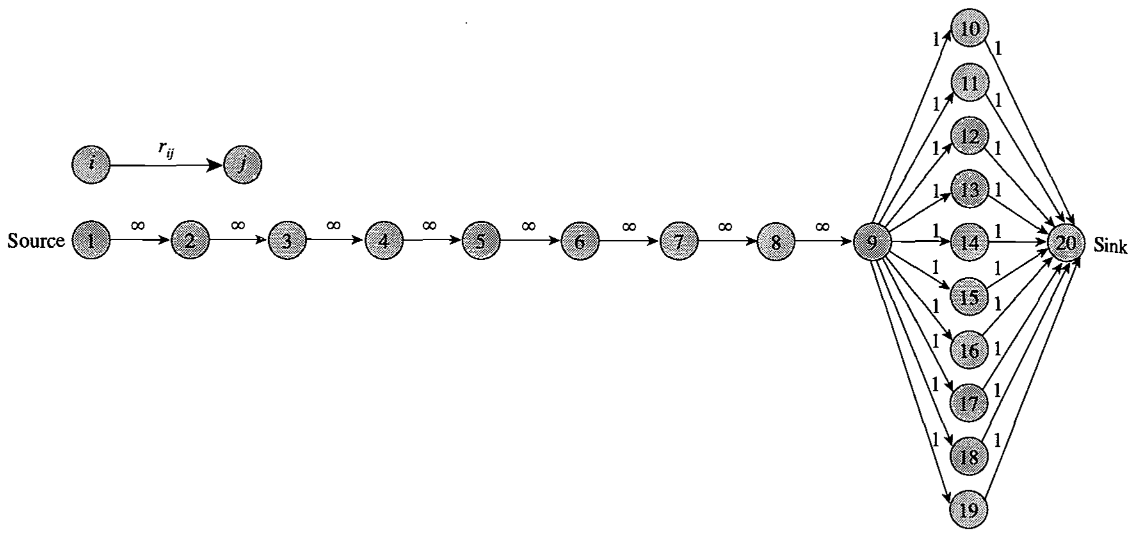


Figure 7.10 Drawback of the augmenting path algorithm.

the node. We refer to any such solution as a *preflow*. More formally, a *preflow* is a function $x: A \to \mathbf{R}$ that satisfies the flow bound constraint (6.1c) and the following relaxation of (6.1b).

$$\sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} \geq 0 \qquad \text{for all } i \in N - \{s, t\}.$$

The preflow-push algorithms maintain a preflow at each intermediate stage. For a given preflow x, we define the *excess* of each node $i \in N$ as

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

In a preflow, $e(i) \geq 0$ for each $i \in N - \{s, t\}$. Moreover, because no arc emanates from node t in the preflow push algorithms, $e(t) \geq 0$ as well. Therefore node s is the only node with negative excess.

We refer to a node with a (strictly) positive excess as an *active* node and adopt the convention that the source and sink nodes are never active. The augmenting path algorithms always maintain feasibility of the solution and strive toward optimality. In contrast, preflow-push algorithms strive to achieve feasibility. In a preflow-push algorithm, the presence of active nodes indicates that the solution is infeasible. Consequently, the basic operation in this algorithm is to select an active node and try to remove its excess by pushing flow to its neighbors. But to which nodes should the flow be sent? Since ultimately we want to send flow to the sink, we push flow to the nodes that are *closer* to sink. As in the shortest augmenting path algorithm, we measure closeness with respect to the current distance labels, so sending flow closer to the sink is equivalent to pushing flow on admissible arcs. Thus we send flow only on admissible arcs. If the active node we are currently considering has no admissible arc, we increase its distance label so that we create at least one admissible arc. The algorithm terminates when the network contains no active node. The preflow-push algorithm uses the subroutines shown in Figure 7.11.

```
procedure preprocess;
begin
    $x := 0$;
    compute the exact distance labels $d(i)$;
    $x_{sj} := u_{sj}$ for each arc $(s, j) \in A(s)$;
    $d(s) := n$;
end;
```

(a)

```
procedure push/relabel(i);
begin
    if the network contains an admissible arc $(i, j)$ then
        push $\delta := \min\{e(i), r_{ij}\}$ units of flow from node $i$ to node $j$
    else replace $d(i)$ by $\min\{d(j) + 1 : (i, j) \in A(i)$ and $r_{ij} > 0\}$;
end;
```

(b)

Figure 7.11 Subroutines of the preflow-push algorithm.

A push of δ units from node i to node j decreases both $e(i)$ and r_{ij} by δ units and increases both $e(j)$ and r_{ji} by δ units. We say that a push of δ units of flow on an arc (i, j) is *saturating* if $\delta = r_{ij}$ and is *nonsaturating* otherwise. A nonsaturating push at node i reduces $e(i)$ to zero. We refer to the process of increasing the distance label of a node as a *relabel* operation. The purpose of the relabel operation is to create at least one admissible arc on which the algorithm can perform further pushes.

The generic version of the preflow-push algorithm (Figure 7.12) combines the subroutines just described.

```
algorithm preflow-push;
begin
    preprocess;
    while the network contains an active node do
    begin
        select an active node $i$;
        push/relabel(i);
    end;
end;
```

Figure 7.12 Generic preflow-push algorithm.

It might be instructive to visualize the generic preflow-push algorithm in terms of a physical network: Arcs represent flexible water pipes, nodes represent joints, and the distance function measures how far nodes are above the ground. In this network we wish to send water from the source to the sink. In addition, we visualize flow in an admissible arc as water flowing downhill. Initially, we move the source node upward, and water flows to its neighbors. In general, water flows downhill towards the sink; however, occasionally flow becomes trapped locally at a node that has no downhill neighbors. At this point we move the node upward, and again water flows downhill toward the sink. Eventually, no more flow can reach the sink. As we continue to move nodes upward, the remaining excess flow eventually flows back toward the source. The algorithm terminates when all the water flows either into the sink or flows back to the source.

The preprocessing operation accomplishes several important tasks. First, it gives each node adjacent to node s a positive excess, so that the algorithm can begin by selecting some node with a positive excess. Second, since the preprocessing operation saturates all the arcs incident to node s, none of these arcs is admissible and setting $d(s) = n$ will satisfy the validity condition (7.2). Third, since $d(s) = n$, Property 7.2 implies that the residual network contains no directed path from node s to node t. Since distance labels are nondecreasing, we also guarantee that in subsequent iterations the residual network will never contain a directed path from node s to node t, and so we will never need to push flow from node s again.

To illustrate the generic preflow-push algorithm, consider the example given in Figure 7.13(a). Figure 7.13(b) specifies the preflow determined by the preprocess operation.

Iteration 1. Suppose that the algorithm selects node 2 for the push/relabel operation. Arc (2, 4) is the only admissible arc and the algorithm performs a push of value $\delta = \min\{e(2), r_{24}\} = \min\{2, 1\} = 1$. This push is saturating. Figure 7.13(c) gives the residual network at this stage.

Iteration 2. Suppose that the algorithm again selects node 2. Since no admissible arc emanates from node 2, the algorithm performs a relabel operation and gives node 2 a new distance label $d(2) = \min\{d(3) + 1, d(1) + 1\} = \min\{2, 5\} = 2$. The new residual network is the same as the one shown in Figure 7.13(c) except that $d(2) = 2$ instead of 1.

Iteration 3. Suppose that this time the algorithm selects node 3. Arc (3, 4) is the only admissible arc emanating from node 3, the algorithm performs a push of value $\delta = \min\{e(3), r_{34}\} = \min\{4, 5\} = 4$. This push is nonsaturating. Figure 7.13(d) gives the residual network at the end of this iteration.

Iteration 4. The algorithm selects node 2 and performs a nonsaturating push of value $\delta = \min\{1, 3\} = 1$, obtaining the residual network given in Figure 7.13(e).

Iteration 5. The algorithm selects node 3 and performs a saturating push of value $\delta = \min\{1, 1\} = 1$ on arc (3, 4), obtaining the residual network given in Figure 7.13(f).

Now the network contains no active node and the algorithm terminates. The maximum flow value in the network is $e(4) = 6$.

Assuming that the generic preflow-push algorithm terminates, we can easily show that it finds a maximum flow. The algorithm terminates when the excess resides at the source or at the sink, implying that the current preflow is a flow. Since $d(s) = n$, the residual network contains no path from the source to the sink. This condition is the termination criterion of the augmenting path algorithm, and the excess residing at the sink is the maximum flow value.

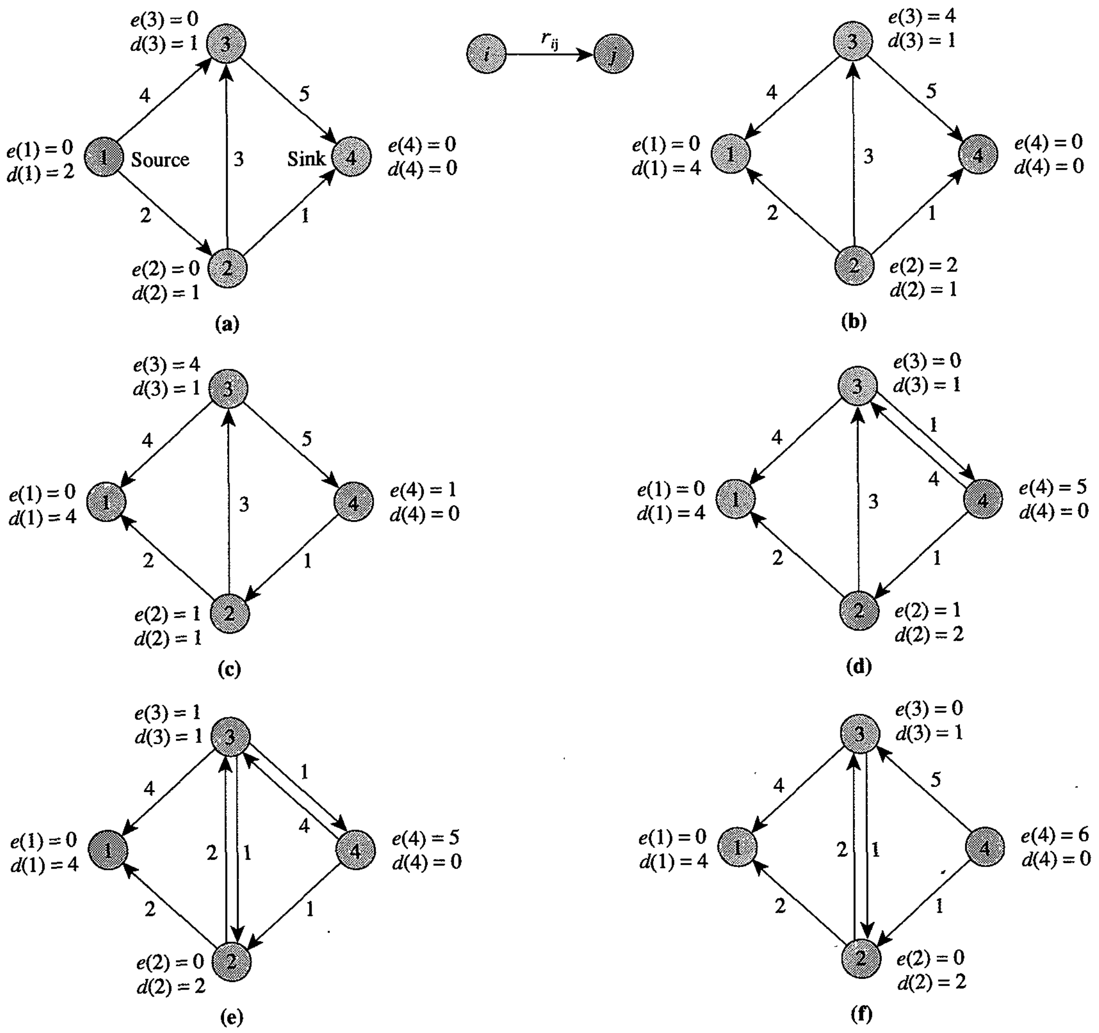


Figure 7.13 Illustrating the generic preflow-push algorithm.

Complexity of the Algorithm

To analyze the complexity of the algorithm, we begin by establishing one important result: distance labels are always valid and do not increase "too many" times. The first of these conclusions follows from Lemma 7.5, because, as in the shortest augmenting path algorithm, the preflow-push algorithm pushes flow only on admissible arcs and relabels a node only when no admissible arc emanates from it. The second conclusion follows from the following lemma.

Lemma 7.11. *At any stage of the preflow-push algorithm, each node i with positive excess is connected to node s by a directed path from node i to node s in the residual network.*

Proof. Notice that for a preflow x, $e(s) \leq 0$ and $e(i) \geq 0$ for all $i \in N - \{s\}$. By the flow decomposition theorem (see Theorem 3.5), we can decompose any preflow x with respect to the original network G into nonnegative flows along (1) paths from node s to node t, (2) paths from node s to active nodes, and (3) flows around directed cycles. Let i be an active node relative to the preflow x in G. The flow decomposition of x must contain a path P from node s to node i, since the paths from node s to node t and the flows around cycles do not contribute to the excess at node i. The residual network contains the reversal of P (P with the orientation of each arc reversed), so a directed path from node i to node s. ◆

This lemma implies that during a relabel operation, the algorithm does not minimize over an empty set.

Lemma 7.12. *For each node $i \in N$, $d(i) < 2n$.*

Proof. The last time the algorithm relabeled node i, the node had a positive excess, so the residual network contained a path P of length at most $n - 2$ from node i to node s. The fact that $d(s) = n$ and that $d(k) \leq d(l) + 1$ for every arc (k, l) in the path P implies that $d(i) \leq d(s) + |P| < 2n$. ◆

Since each time the algorithm relabels node i, $d(i)$ increases by at least 1 unit, we have established the following result.

Lemma 7.13. *Each distance label increases at most $2n$ times. Consequently, the total number of relabel operations is at most $2n^2$.* ◆

Lemma 7.14. *The algorithm performs at most nm saturating pushes.*

Proof. This result follows directly from Lemmas 7.12 and 7.8. ◆

In view of Lemma 7.7, Lemma 7.13 implies that the total time needed to identify admissible arcs and to perform relabel operations is $O(nm)$. We next count the number of nonsaturating pushes performed by the algorithm.

Lemma 7.15. *The generic preflow-push algorithm performs $O(n^2m)$ nonsaturating pushes.*

Proof. We prove the lemma using an argument based on potential functions (see Section 3.2). Let I denote the set of active nodes. Consider the potential function $\Phi = \sum_{i \in I} d(i)$. Since $|I| < n$, and $d(i) < 2n$ for all $i \in I$, the initial value of Φ (after the preprocess operation) is at most $2n^2$. At the termination of the algorithm, Φ is zero. During the push/relabel(i) operation, one of the following two cases must apply:

Case 1. The algorithm is unable to find an admissible arc along which it can push flow. In this case the distance label of node i increases by $\epsilon \geq 1$ units. This operation increases Φ by at most ϵ units. Since the total increase in $d(i)$ for each node i throughout the execution of the algorithm is bounded by $2n$, the total increase in Φ due to increases in distance labels is bounded by $2n^2$.

Case 2. The algorithm is able to identify an arc on which it can push flow, so it performs a saturating push or a nonsaturating push. A saturating push on arc (i, j) might create a new excess at node j, thereby increasing the number of active nodes by 1, and increasing Φ by $d(j)$, which could be as much as $2n$ per saturating push, and so $2n^2m$ over all saturating pushes. Next note that a nonsaturating push on arc (i, j) does not increase $|I|$. The nonsaturating push will decrease Φ by $d(i)$ since i becomes inactive, but it simultaneously increases Φ by $d(j) = d(i) - 1$ if the push causes node j to become active, the total decrease in Φ being of value 1. If node j was active before the push, Φ decreases by an amount $d(i)$. Consequently, net decrease in Φ is at least 1 unit per nonsaturating push.

We summarize these facts. The initial value of Φ is at most $2n^2$ and the maximum possible increase in Φ is $2n^2 + 2n^2m$. Each nonsaturating push decreases Φ by at least 1 unit and Φ always remains nonnegative. Consequently, the algorithm can perform at most $2n^2 + 2n^2 + 2n^2m = O(n^2m)$ nonsaturating pushes, proving the lemma. ◆

Finally, we indicate how the algorithm keeps track of active nodes for the push/relabel operations. The algorithm maintains a set LIST of active nodes. It adds to LIST those nodes that become active following a push and are not already in LIST, and deletes from LIST nodes that become inactive following a nonsaturating push. Several data structures (e.g., doubly linked lists) are available for storing LIST so that the algorithm can add, delete, or select elements from it in $O(1)$ time. Consequently, it is easy to implement the preflow-push algorithm in $O(n^2m)$ time. We have thus established the following theorem.

Theorem 7.16. *The generic preflow-push algorithm runs in $O(n^2m)$ time.* ◆

Several modifications to the generic preflow-push algorithm might improve its empirical performance. We define a maximum preflow as a preflow with the maximum possible flow into the sink. As stated, the generic preflow-push algorithm performs push/relabel operations at active nodes until all the excess reaches the sink node or returns to the source node. Typically, the algorithm establishes a maximum preflow long before it establishes a maximum flow; the subsequent push/relabel operations increase the distance labels of the active nodes until they are sufficiently higher than n so they can push their excesses back to the source node (whose distance label is n). One possible modification in the preflow-push algorithm is to maintain a set N' of nodes that satisfy the property that the residual network contains no path from a node in N' to the sink node t. Initially, $N' = \{s\}$ and, subsequently, whenever the distance label of a node is greater than or equal to n, we add it to N'. Further, we do not perform push/relabel operations for nodes in N' and terminate the algorithm when all nodes in $N - N'$ are inactive. At termination, the current preflow x is also an optimal preflow. At this point we convert the maximum preflow x into a maximum flow using any of the methods described in Exercise 7.11. Empirical tests have found that this two-phase approach often substantially reduces the running times of preflow push algorithms.

One sufficient condition for adding a node j to N' is $d(j) \geq n$. Unfortunately, this simple approach is not very effective and does not substantially reduce the running time of the algorithm. Another approach is to occasionally perform a reverse breadth-first search of the residual network to obtain exact distance labels and add all those nodes to N' that do not have any directed path to the sink. Performing this search occasionally, that is, after αn relabel operations for some constant α, does not effect the worst-case complexity of the preflow-push algorithm (why?) but improves the empirical behavior of the algorithm substantially.

A third approach is to let numb(k) denote the number of nodes whose distance label is k. As discussed in Section 7.4, we can update the array numb($\cdot$) in $O(1)$ steps per relabel operation. Moreover, whenever numb(k') = 0 for some k', any node j with $d(j) > k'$ is disconnected from the set of nodes i with $d(i) < k'$ in the residual network. At this point, we can increase the distance labels of each of these nodes to n and the distance labels will still be valid (why?). Equivalently, we can add any node j with $d(j) > k'$ to the set N'. The array numb($\cdot$) is easy to implement, and its use is quite effective in practice.

Specific Implementations of Generic Preflow-Push Algorithm

The running time of the generic preflow-push algorithm is comparable to the bound of the shortest augmenting path algorithm. However, the preflow-push algorithm has several nice features: in particular, its flexibility and its potential for further improvements. By specifying different rules for selecting active nodes for the push/relabel operations, we can derive many different algorithms, each with different worst-case complexity than the generic version of the algorithm. The bottleneck operation in the generic preflow-push algorithm is the number of nonsaturating pushes and many specific rules for examining active nodes can produce substantial reductions in the number of nonsaturating pushes. We consider the following three implementations.

1. *FIFO preflow-push algorithm.* This algorithm examines the active nodes in the first-in, first-out (FIFO) order. We shall show that this algorithm runs in $O(n^3)$ time.
2. *Highest-label preflow-push algorithm.* This algorithm always pushes from an active node with the highest value of the distance label. We shall show that this algorithm runs in $O(n^2m^{1/2})$ time. Observe that this time is better than $O(n^3)$ for all problem densities.
3. *Excess scaling algorithm.* This algorithm pushes flow from a node with sufficiently large excess to a node with sufficiently small excess. We shall show that the excess scaling algorithm runs in $O(nm + n^2 \log U)$ time. For problems that satisfy the similarity assumption (see Section 3.2), this time bound is better than that of the two preceding algorithms.

We might note that the time bounds for all these preflow-push algorithms are tight (except the excess scaling algorithm); that is, for some classes of networks the generic preflow-push algorithm, the FIFO algorithm, and the highest-label preflow-push algorithms do perform as many computations as indicated by their worst-case time bounds. These examples show that we cannot improve the time bounds of these algorithms by a more clever analysis.

7.7 FIFO PREFLOW-PUSH ALGORITHM

Before we describe the FIFO implementation of the preflow-push algorithm, we define the concept of a *node examination*. In an iteration, the generic preflow-push algorithm selects a node, say node *i*, and performs a saturating push or a nonsaturating push, or relabels the node. If the algorithm performs a saturating push, then node *i* might still be active, but it is not mandatory for the algorithm to select this node again in the next iteration. The algorithm might select another node for the next push/relabel operation. However, it is easy to incorporate the rule that whenever the algorithm selects an active node, it keeps pushing flow from that node until either the node's excess becomes zero or the algorithm relabels the node. Consequently, the algorithm might perform several saturating pushes followed either by a nonsaturating push or a relabel operation. We refer to this sequence of operations as a node examination. We shall henceforth assume that every preflow-push algorithm adopts this rule for selecting nodes for the push/relabel operation.

The FIFO preflow-push algorithm examines active nodes in the FIFO order. The algorithm maintains the set LIST as a queue. It selects a node *i* from the front of LIST, performs pushes from this node, and adds newly active nodes to the rear of LIST. The algorithm examines node *i* until either it becomes inactive or it is relabeled. In the latter case, we add node *i* to the rear of the queue. The algorithm terminates when the queue of active nodes is empty.

We illustrate the FIFO preflow-push algorithm using the example shown in Figure 7.14(a). The preprocess operation creates an excess of 10 units at each of

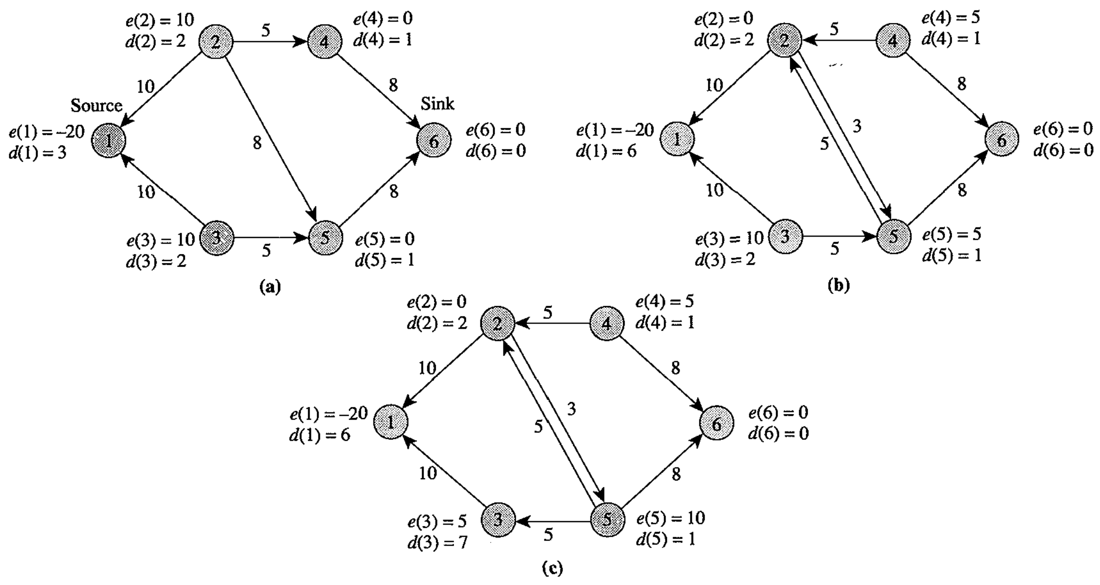


Figure 7.14 Illustrating the FIFO preflow-push algorithm.

the nodes 2 and 3. Suppose that the queue of active nodes at this stage is LIST = {2, 3}. The algorithm removes node 2 from the queue and examines it. Suppose that it performs a saturating push of 5 units on arc (2, 4) and a nonsaturating push of 5 units on arc (2, 5) [see Figure 7.14(b)]. As a result of these pushes, nodes 4 and 5 become active and we add these nodes to the queue in this order, obtaining LIST = {3, 4, 5}. The algorithm next removes node 3 from the queue. While examining node 3, the algorithm performs a saturating push of 5 units on arc (3, 5), followed by a relabel operation of node 3 [see Figure 7.14(c)]. The algorithm adds node 3 to the queue, obtaining LIST = {4, 5, 3}. We encourage the reader to complete the solution of this example.

To analyze the worst-case complexity of the FIFO preflow-push algorithm, we partition the total number of node examinations into different phases. The first phase consists of node examinations for those nodes that become active during the preprocess operation. The second phase consists of the node examinations of all the nodes that are in the queue after the algorithm has examined the nodes in the first phase. Similarly, the third phase consists of the node examinations of all the nodes that are in the queue after the algorithm has examined the nodes in the second phase, and so on. For example, in the preceding illustration, the first phase consists of the node examinations of the set {2, 3}, and the second phase consists of the node examinations of the set {4, 5, 3}. Observe that the algorithm examines any node at most once during a phase.

We will now show that the algorithm performs at most $2n^2 + n$ phases. Each phase examines any node at most once and each node examination performs at most one nonsaturating push. Therefore, a bound of $2n^2 + n$ on the total number of phases would imply a bound of $O(n^3)$ on the number of nonsaturating pushes. This result would also imply that the FIFO preflow-push algorithm runs in $O(n^3)$ time because the bottleneck operation in the generic preflow-push algorithm is the number of nonsaturating pushes.

To bound the number of phases in the algorithm, we consider the total change in the potential function $\Phi = \max\{d(i) : i \text{ is active}\}$ over an entire phase. By the "total change" we mean the difference between the initial and final values of the potential function during a phase. We consider two cases.

Case 1. The algorithm performs at least one relabel operation during a phase. Then Φ might increase by as much as the maximum increase in any distance label. Lemma 7.13 implies that the total increase in Φ over all the phases is at most $2n^2$.

Case 2. The algorithm performs no relabel operation during a phase. In this case the excess of every node that was active at the beginning of the phase moves to nodes with smaller distance labels. Consequently, Φ decreases by at least 1 unit.

Combining Cases 1 and 2, we find that the total number of phases is at most $2n^2 + n$; the second term corresponds to the initial value of Φ, which could be at most n. We have thus proved the following theorem.

Theorem 7.17. *The FIFO preflow-push algorithm runs in $O(n^3)$ time.* ◆

7.8 HIGHEST-LABEL PREFLOW-PUSH ALGORITHM

The highest-label preflow-push algorithm always pushes flow from an active node with the highest distance label. It is easy to develop an $O(n^3)$ bound on the number of nonsaturating pushes for this algorithm. Let $h^* = \max\{d(i): i \text{ is active}\}$. The algorithm first examines nodes with distance labels equal to h^* and pushes flow to nodes with distance labels equal to $h^* - 1$, and these nodes, in turn, push flow to nodes with distance labels equal to $h^* - 2$, and so on, until either the algorithm relabels a node or it has exhausted all the active nodes. When it has relabeled a node, the algorithm repeats the same process. Note that if the algorithm does not relabel any node during n consecutive node examinations, all the excess reaches the sink (or the source) and the algorithm terminates. Since the algorithm performs at most $2n^2$ relabel operations (by Lemma 7.13), we immediately obtain a bound of $O(n^3)$ on the number of node examinations. Since each node examination entails at most one nonsaturating push, the highest-label preflow-push algorithm performs $O(n^3)$ nonsaturating pushes. (In Exercise 7.20 we consider a potential function argument that gives the same bound on the number of nonsaturating pushes.)

The preceding discussion is missing one important detail: How do we select a node with the highest distance label without expending too much effort? We use the following data structure. For each $k = 1, 2, \ldots, 2n - 1$, we maintain the list

$$\text{LIST}(k) = \{i : i \text{ is active and } d(i) = k\},$$

in the form of either linked stacks or linked queues (see Appendix A). We define a variable *level* that is an upper bound on the highest value of k for which LIST(k) is nonempty. To determine a node with the highest distance label, we examine the lists LIST(level), LIST(level-1), . . . , until we find a nonempty list, say LIST(p). We set level equal to p and select any node in LIST(p). Moreover, if the distance label of a node increases while the algorithm is examining it, we set level equal to the new distance label of the node. Observe that the total increase in level is at most $2n^2$ (from Lemma 7.13), so the total decrease is at most $2n^2 + n$. Consequently, scanning the lists LIST(level), LIST(level-1), . . . , in order to find the first nonempty list is not a bottleneck operation.

The highest-label preflow-push algorithm is currently the most efficient method for solving the maximum flow problem in practice because it performs the least number of nonsaturating pushes. To illustrate intuitively why the algorithm performs so well in practice, we consider the maximum flow problem given in Figure 7.15(a). The preprocess operation creates an excess of 1 unit at each node $2, 3, \ldots, n - 1$ [see Figure 7.15(b)]. The highest-label preflow-push algorithm examines nodes $2, 3, \ldots, n - 1$, in this order and pushes all the excess to the sink node. In contrast, the FIFO preflow-push algorithm might perform many more pushes. Suppose that at the end of the preprocess operation, the queue of active nodes is LIST $= \{n - 1, n - 2, \ldots, 3, 2\}$. Then the algorithm would examine each of these nodes in the first phase and would obtain the solution depicted in Figure 7.15(c). At this point, LIST $= \{n - 1, n - 2, \ldots, 4, 3\}$. It is easy to show that overall the algorithm would perform $n - 2$ phases and use $(n - 2) + (n - 3) + \ldots + 1 = \Omega(n^2)$ nonsaturating pushes.

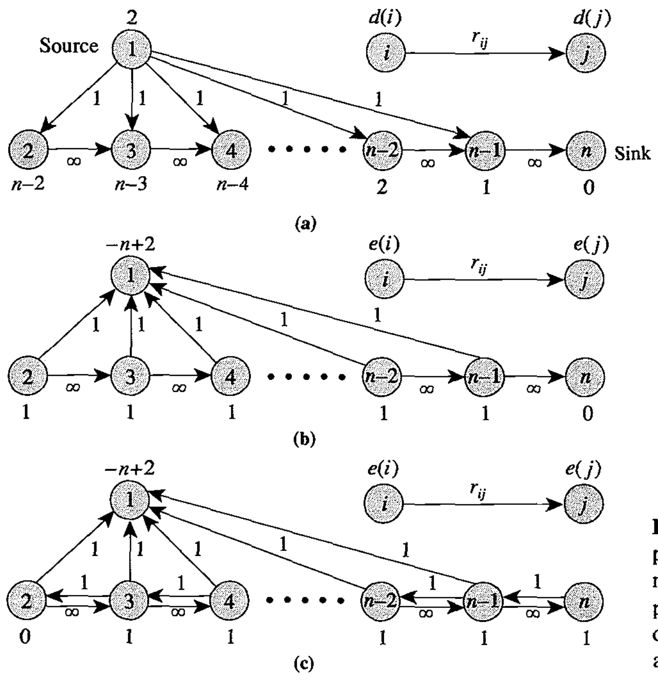


Figure 7.15 Bad example for the FIFO preflow-push algorithm: (a) initial residual network; (b) network after the preprocess operation; (c) network after one phase of the FIFO preflow-push algorithm.

Although the preceding example is rather extreme, it does illustrate the advantage in pushing flows from active nodes with the highest distance label. In our example the FIFO algorithm selects an excess and pushes it all the way to the sink. Then it selects another excess and pushes it to the sink, and repeats this process until no node contains any more excess. On the other hand, the highest-label preflow-push algorithm starts at the highest level and pushes all the excess at this level to the next lower level and repeats this process. As the algorithm examines nodes with lower and lower distance labels, it accumulates the excesses and pushes this accumulated excess toward the sink. Consequently, the highest-label preflow-push algorithm avoids repetitive pushes on arcs carrying a small amount of flow.

This nice feature of the highest-label preflow-push algorithm also translates into a tighter bound on the number of nonsaturating pushes. The bound of $O(n^3)$ on the number of nonsaturating pushes performed by the algorithm is rather loose and can be improved by a more clever analysis. We now show that the algorithm in fact performs $O(n^2m^{1/2})$ nonsaturating pushes. The proof of this theorem is somewhat complex and the reader can skip it without any loss of continuity.

At every state of the preflow-push algorithm, each node other than the sink has at most one current arc which, by definition, must be admissible. We denote this collection of current arcs by the set F. The set F has at most $n - 1$ arcs, has at most one outgoing arc per node, and does not contain any cycle (why?). These results imply that F defines a *forest*, which we subsequently refer to as the *current forest*. Figure 7.16 gives an example of a current forest. Notice that each tree in the forest is a rooted tree, the root being a node with no outgoing arc.

Before continuing, let us introduce some additional notation. For any node $i \in N$, we let $D(i)$ denote the set of descendants of that node in F (we refer the

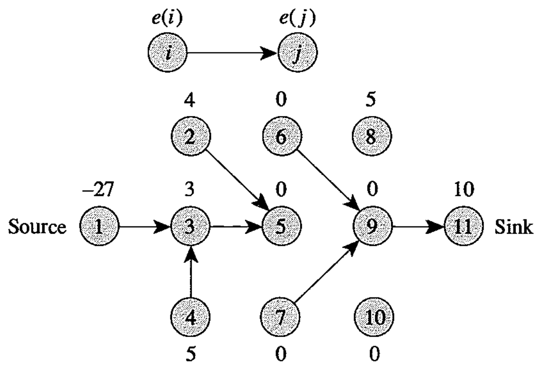


Figure 7.16 Example of a current-forest.

reader to Section 2.2 for the definition of descendants in a rooted tree). For example, in Figure 7.16, $D(1) = \{1\}$, $D(2) = \{2\}$, $D(3) = \{1, 3, 4\}$, $D(4) = \{4\}$, and $D(5) = \{1, 2, 3, 4, 5\}$. Notice that distance label of the descendants of any node i will be higher than $d(i)$. We refer to an active node with no active descendants (other than itself) as a *maximal active node.* In Figure 7.16 the nodes 2, 4, and 8 are the only maximal active nodes. Let H denote the set of maximal active nodes. Notice that two maximal active nodes have distinct descendants. Also notice that the highest-label preflow-push algorithm always pushes flow from a maximal active node.

We obtain the time bound of $O(n^2m^{1/2})$ for the highest-label preflow-push algorithm using a potential function argument. The argument relies on a parameter K, whose optimal value we select later. Our potential function is $\Phi = \sum_{i \in H} \Phi(i)$, with $\Phi(i)$ defined as $\Phi(i) = \max\{0, K + 1 - |D(i)|\}$. Observe that for any node i, $\Phi(i)$ is at most K [because $|D(i)| \geq 1$]. Also observe that Φ changes whenever the set H of maximal active nodes changes or $|D(i)|$ changes for a maximal active node i.

We now study the effect of various operations performed by the preflow-push algorithm on the potential function Φ. As the algorithm proceeds, it changes the set of current arcs, performs saturating and nonsaturating pushes; and relabels nodes. All these operations have an effect on the value of Φ. By observing the consequence of all these operations on Φ, we will obtain a bound on the number of nonsaturating pushes.

First, consider a nonsaturating push on an arc (i, j) emanating from a maximal active node i. Notice that a nonsaturating push takes place on a current arc and does not change the current forest; it simply moves the excess from node i to node j [see Figure 7.17(a) for a nonsaturating push on the arc (3, 4)]. As a result of the push, node i becomes inactive and node j might become a new maximal active node. Since $|D(j)| > |D(i)|$, this push decreases $\Phi(i) + \Phi(j)$ by at least 1 unit if $|D(i)| \leq K$ and does not change $\Phi(i) + \Phi(j)$ otherwise.

Now consider a saturating push on the arc (i, j) emanating from a maximal active node i. As a result of the push, arc (i, j) becomes inadmissible and drops out of the current forest [see Figure 7.17(b) for a saturating push on the arc (1, 3)]. Node i remains a maximal active node and node j might also become a maximal active node. Consequently, this operation might increase Φ by upto K units.

Next consider the relabeling of a maximal active node i. We relabel a node when it has no admissible arc; therefore, no current arc emanates from this node.

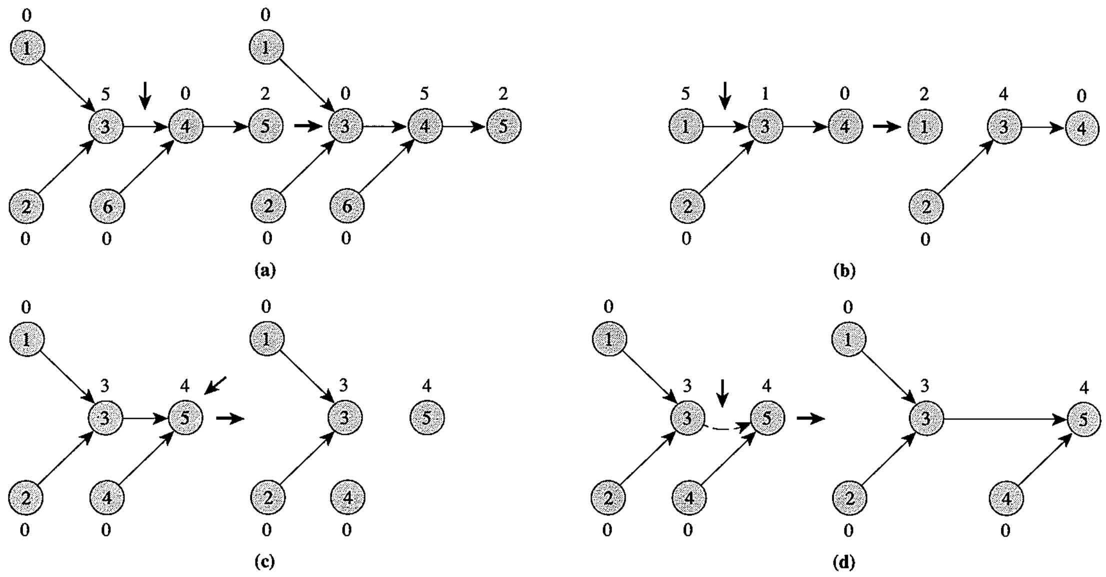


Figure 7.17 (a) Nonsaturating push on arc (d, 4); (b) saturating push on arc (1, 3); (c) relabel of node 5; (d) addition of the arc (3, 5) to the forest.

As a consequence, node *i* must be a root node in the current forest. Moreover, since node *i* is a maximal active node, none of its proper descendants can be active. After the algorithm has relabeled node *i*, all incoming arcs at node *i* become inadmissible; therefore, all the current arcs entering node *i* will no longer belong to the current forest [see Figure 7.17(c)]. Clearly, this change cannot create any new maximal active nodes. The relabel operation, however, decreases the number of descendants of node *i* to one. Consequently, $\Phi(i)$ can increase by at most K.

Finally, consider the introduction of new current arcs in the current forest. The addition of new arcs to the forest does not create any new maximal active nodes. It might, in fact, remove some maximal active nodes and increase the number of descendants of some nodes [see Figure 7.17(d)]. In both cases the potential Φ does not increase. We summarize the preceding discussion in the form of the following property.

Property 7.18

(a) *A nonsaturating push from a maximal active node i does not increase* Φ; *it decreases* Φ *by at least 1 unit if* $|D(i)| \leq K$.
(b) *A saturating push from a maximal active node i can increase* Φ *by at most* K *units.*
(c) *The relabeling of a maximal active node i can increase* Φ *by at most* K *units.*
(d) *Introducing current arcs does not increase* Φ.

For the purpose of worst-case analysis,. we define the concept of phases. A *phase* consists of the sequence of pushes between two consecutive relabel operations. Lemma 7.13 implies that the algorithm contains $O(n^2)$ phases. We call a phase *cheap* if it performs at most $2n/K$ nonsaturating pushes, and *expensive* otherwise.

Clearly, the number of nonsaturating pushes in cheap phases is at most $O(n^2 \cdot 2n/K) = O(n^3/K)$. To obtain a bound on the nonsaturating pushes in expensive phases, we use an argument based on the potential function Φ.

By definition, an expensive phase performs at least $2n/K$ nonsaturating pushes. Since the network can contain at most n/K nodes with K descendants or more, at least n/K nonsaturating pushes must be from nodes with fewer than K descendants. The highest-label preflow-push algorithm always performs push/relabel operation on a maximal active node; consequently, Property 7.18 applies. Property 7.18(a) implies that each of these nonsaturating pushes produces a decrease in Φ of at least 1. So Properties 7.18(b) and (c) imply that the total increase in Φ due to saturating pushes and relabels is at most $O(nmK)$. Therefore, the algorithm can perform $O(nmK)$ nonsaturating pushes in expensive phases.

To summarize this discussion, we note that cheap phases perform $O(n^3/K)$ nonsaturating pushes and expensive phases perform $O(nmK)$ nonsaturating pushes. We obtain the optimal value of K by balancing both terms (see Section 3.2), that is, when both the terms are equal: $n^3/K = nmK$ or $K = n/m^{1/2}$. For this value of K, the number of nonsaturating pushes is $O(n^2m^{1/2})$. We have thus established the following result.

Theorem 7.19. *The highest-label preflow-push algorithm performs $O(n^2m^{1/2})$ nonsaturating pushes and runs in the same time.* ◆

7.9 EXCESS SCALING ALGORITHM

The generic preflow-push algorithm allows flow at each intermediate step to violate the mass balance equations. By pushing flows from active nodes, the algorithm attempts to satisfy the mass balance equations. The function $e_{\max} = \max\{e(i): i$ is an active node$\}$ provides one measure of the infeasibility of a preflow. Note that during the execution of the generic algorithm, we would observe no particular pattern in the values of $e_{\max}$, except that $e_{\max}$ eventually decreases to value 0. In this section we develop an *excess scaling technique* that systematically reduces the value of $e_{\max}$ to 0.

The excess scaling algorithm is similar to the capacity scaling algorithm we discussed in Section 7.3. Recall that the generic augmenting path algorithm performs $O(nU)$ augmentations and the capacity scaling algorithm reduces this number to $O(m \log U)$ by assuring that each augmentation carries a "sufficiently large" amount of flow. Similarly, in the generic preflow-push algorithm, nonsaturating pushes carrying small amount of flow bottleneck the algorithm in theory. The excess scaling algorithm assures that each nonsaturating push carries a "sufficiently large" amount of flow and so the number of nonsaturating pushes is "sufficiently small."

Let Δ denote an upper bound on $e_{\max}$; we refer to this bound as the *excess dominator*. We refer to a node with $e(i) \geq \Delta/2 \geq e_{\max}/2$ as a node with *large excess*, and as a node with *small excess* otherwise. The excess scaling algorithm always pushes flow from a node with a large excess. This choice assures that during nonsaturating pushes, the algorithm sends relatively large excess closer to the sink.

The excess scaling algorithm also does not allow the maximum excess to increase beyond Δ. This algorithmic strategy might prove to be useful for the following reason. Suppose that several nodes send flow to a single node j, creating a very large excess. It is likely that node j cannot send the accumulated flow closer to the sink, and thus the algorithm will need to increase its distance label and return much of its excess back to the nodes it came from. Thus pushing too much flow to any node is also likely to be a wasted effort.

The two conditions we have discussed—that each nonsaturating push must carry at least $\Delta/2$ units of flow and that no excess should exceed Δ—imply that we need to select the active nodes for push/relabel operations carefully. The following selection rule is one that assures that we achieve these objectives.

Node Selection Rule. *Among all nodes with a large excess, select a node with the smallest distance label (breaking ties arbitrarily).*

We are now in a position to give, in Figure 7.18, a formal description of the excess scaling algorithm.

The excess scaling algorithm uses the same push/relabel(i) operation as the generic preflow-push algorithm, but with one slight difference. Instead of pushing $\delta = \min\{e(i), r_{ij}\}$ units of flow, it pushes $\delta = \min\{e(i), r_{ij}, \Delta - e(j)\}$ units. This change ensures that the algorithm permits no excess to exceed Δ.

The algorithm performs a number of scaling phases with the value of the excess dominator Δ decreasing from phase to phase. We refer to a specific scaling phase with a particular value of Δ as a *Δ-scaling phase*. Initially, $\Delta = 2^{\lceil \log U \rceil}$. Since the logarithm is of base 2, $U \le \Delta \le 2U$. During the Δ-scaling phase, $\Delta/2 < e_{\max} \le \Delta$; the value of $e_{\max}$ might increase or decrease during the phase. When $e_{\max} \le \Delta/2$, we begin a new scaling phase. After the algorithm has performed $\lceil \log U \rceil + 1$ scaling phases, $e_{\max}$ decreases to value 0 and we obtain the maximum flow.

Lemma 7.20. *The algorithm satisfies the following two conditions:*
(a) *Each nonsaturating push sends at least $\Delta/2$ units of flow.*
(b) *No excess ever exceeds Δ.*

Proof. Consider a nonsaturating push on arc (i, j). Since arc (i, j) is admissible, $d(j) < d(i)$. Moreover, since node i is a node with the smallest distance label among all nodes with a large excess, $e(i) \ge \Delta/2$ and $e(j) < \Delta/2$. Since this push is non-

```
algorithm excess scaling;
begin
    preprocess;
    $\Delta := 2^{\lceil \log U \rceil}$;
    while $\Delta \ge 1$ do
    begin ($\Delta$-scaling phase)
        while the network contains a node $i$ with a large excess do
            begin
                among all nodes with a large excess, select a node $i$ with
                the smallest distance label;
                perform push/relabel($i$) while ensuring that no node excess exceeds $\Delta$;
            end;
        $\Delta := \Delta/2$;
    end;
end;
```

Figure 7.18 Excess scaling algorithm.

saturating, it sends $\min\{e(i), \Delta - e(j)\} \geq \Delta/2$ units of flow, proving the first part of the lemma. This push operation increases the excess of only node j. The new excess of node j is $e(j) + \min\{e(i), \Delta - e(j)\} \leq e(j) + \{\Delta - e(j)\} \leq \Delta$. So all the node excesses remain less than or equal to Δ. This proves the second part of the lemma. ◆

Lemma 7.21. *The excess scaling algorithm performs $O(n^2)$ nonsaturating pushes per scaling phase and $O(n^2 \log U)$ pushes in total.*

Proof. Consider the potential function $\Phi = \sum_{i \in N} e(i)d(i)/\Delta$. Using this potential function, we will establish the first assertion of the lemma. Since the algorithm performs $O(\log U)$ scaling phases, the second assertion is a consequence of the first. The initial value of Φ at the beginning of the Δ-scaling phase is bounded by $2n^2$ because $e(i)$ is bounded by Δ and $d(i)$ is bounded by $2n$. During the push/relabel(i) operation, one of the following two cases must apply:

Case 1. The algorithm is unable to find an admissible arc along which it can push flow. In this case the distance label of node i increases by $\epsilon \geq 1$ units. This relabeling operation increases Φ by at most ϵ units because $e(i) \leq \Delta$. Since for each i the total increase in $d(i)$ throughout the running of the algorithm is bounded by $2n$ (by Lemma 7.13), the total increase in Φ due to the relabeling of nodes is bounded by $2n^2$ in the Δ-scaling phase (actually, the increase in Φ due to node relabelings is at most $2n^2$ over *all* scaling phases).

Case 2. The algorithm is able to identify an arc on which it can push flow, so it performs either a saturating or a nonsaturating push. In either case, Φ decreases. A nonsaturating push on arc (i, j) sends at least $\Delta/2$ units of flow from node i to node j and since $d(j) = d(i) - 1$, after this operation decreases Φ by at least $\frac{1}{2}$ unit. Since the initial value of Φ at the beginning of a Δ-scaling phase is at most $2n^2$ and the increases in Φ during this scaling phase sum to at most $2n^2$ (from Case 1), the number of nonsaturating pushes is bounded by $8n^2$. ◆

This lemma implies a bound of $O(nm + n^2 \log U)$ on the excess scaling algorithm since we have already seen that all the other operations—such as saturating pushes, relabel operations, and finding admissible arcs—require $O(nm)$ time. Up to this point we have ignored the method needed to identify a node with the minimum distance label among nodes with excess more than $\Delta/2$. Making this identification is easy if we use a scheme similar to the one used in the highest-label preflow-push algorithm in Section 7.8 to find a node with the highest distance label. We maintain the lists *LIST(k)* $= \{i \in N: e(i) > \Delta/2 \text{ and } d(i) = k\}$, and a variable *level* that is a lower bound on the smallest index k for which LIST(k) is nonempty. We identify the lowest-indexed nonempty list by starting at LIST(level) and sequentially scanning the higher-indexed lists. We leave as an exercise to show that the overall effort needed to scan the lists is bounded by the number of pushes performed by the algorithm plus $O(n \log U)$, so these computations are not a bottleneck operation. With this observation we can summarize our discussion as follows.

Theorem 7.22. *The excess scaling algorithm runs in $O(nm + n^2 \log U)$ time.* ◆

Algorithm	Running time	Features
Labeling algorithm	$O(nmU)$	1. Maintains a feasible flow and augments flows along directed paths in the residual network from node s to node t. 2. Easy to implement and very flexible. 3. Running time is pseudopolynomial: the algorithm is not very efficient in practice.
Capacity scaling algorithm	$O(nm \log U)$	1. A special implementation of the labeling algorithm. 2. Augments flows along paths from node s to node t with sufficiently large residual capacity. 3. Unlikely to be efficient in practice.
Successive shortest path algorithm	$O(n^2m)$	1. Another special implementation of the labeling algorithm. 2. Augments flows along shortest directed paths from node s to node t in the residual network. 3. Uses distance labels to identify shortest paths from node s to node t. 4. Relatively easy to implement and very efficient in practice.
Generic preflow-push algorithm	$O(n^2m)$	1. Maintains a pseudoflow; performs push/relabel operations at active nodes. 2. Very flexible; can examine active nodes in any order. 3. Relatively difficult to implement because an efficient implementation requires the use of several heuristics.
FIFO preflow-push algorithm	$O(n^3)$	1. A special implementation of the generic preflow-push algorithm. 2. Examines active nodes in the FIFO order. 3. Very efficient in practice.
Highest-label preflow-push algorithm	$O(n^2\sqrt{m})$	1. Another special implementation of the generic preflow-push algorithm. 2. Examines active nodes with the highest distance label. 3. Possibly the most efficient maximum flow algorithm in practice.
Excess scaling algorithm	$O(nm + n^2 \log U)$	1. A special implementation of the generic preflow-push algorithm. 2. Performs push/relabel operations at nodes with sufficiently large excesses and, among these nodes, selects a node with the smallest distance label. 3. Achieves an excellent running time without using sophisticated data structures.

Figure 7.19 Summary of maximum flow algorithms.

7.10 SUMMARY

Building on the labeling algorithm described in Chapter 6, in this chapter we described several polynomial-time algorithms for the maximum flow problem. The labeling algorithm can perform as many as nU augmentations because each augmentation might carry a small amount of flow. We studied two natural strategies for reducing the number of augmentations and thus for improving the algorithm's running time; these strategies lead to the capacity scaling algorithm and the shortest augmenting path algorithm. One inherent drawback of these augmenting path algorithms is the computationally expensive operation of sending flows along paths. These algorithms might repeatedly augment flows along common path segments. The preflow-push algorithms that we described next overcome this drawback; we can conceive of them as sending flows along several paths simultaneously. In our development we considered both a generic implementation and several specific implementations of the preflow-push algorithm. The FIFO and highest-label preflow-push algorithms choose the nodes for pushing/relabeling in a specific order. The excess scaling algorithm ensures that the push operations, and subsequent augmentations, do not carry small amounts of flow (with "small" defined dynamically throughout the algorithm). Figure 7.19 summarizes the running times and basic features of these algorithms.

REFERENCE NOTES

The maximum flow problem is distinguished by the long succession of research contributions that have improved on the worst-case complexity of the best known algorithms. Indeed, no other network flow problem has witnessed as many incremental improvements. The following discussion provides a brief survey of selective improvements; Ahuja, Magnanti, and Orlin [1989, 1991] give a more complete survey of the developments in this field.

The labeling algorithm of Ford and Fulkerson [1956a] runs in pseudopolynomial time. Edmonds and Karp [1972] suggested two polynomial-time implementations of this algorithm. The first implementation, which augments flow along paths with the maximum residual capacity, performs $O(m \log U)$ iterations. The second implementation, which augments flow along shortest paths, performs $O(nm)$ iterations and runs in $O(nm^2)$ time. Independently, Dinic [1970] introduced a concept of shortest path networks (in number of arcs), called *layered networks*, and obtained an $O(n^2m)$-time algorithm. Until this point all maximum flow algorithms were augmenting path algorithms. Karzanov [1974] introduced the first preflow-push algorithm on layered networks; he obtained an $O(n^3)$ algorithm. Shiloach and Vishkin [1982] described another $O(n^3)$ preflow-push algorithm for the maximum flow problem, which is a precursor of the FIFO preflow-push algorithm that we described in Section 7.7.

The capacity scaling described in Section 7.3 is due to Ahuja and Orlin [1991]; this algorithm is similar to the bit-scaling algorithm due to Gabow [1985] that we describe in Exercise 7.19. The shortest augmenting path algorithm described in Section 7.4 is also due to Ahuja and Orlin [1991]; this algorithm can be regarded as a variant of Dinic's [1970] algorithm and uses distance labels instead of layered networks.

Researchers obtained further improvements in the running times of the maximum flow algorithms by using distance labels instead of layered networks. Goldberg [1985] first introduced distance labels; by incorporating them in the algorithm of Shiloach and Vishkin [1982], he obtained the $O(n^3)$-time FIFO implementation that we described in Section 7.7. The generic preflow-push algorithm and its highest-label preflow-push implementation that we described in Sections 7.8 are due to Goldberg and Tarjan [1986]. Using a dynamic tree data structure developed by Sleator and Tarjan [1983], Goldberg and Tarjan [1986] improved the running time of the FIFO implementation to $O(nm \log(n^2/m))$. Using a clever analysis, Cheriyan and Maheshwari [1989] show that the highest-label preflow-push algorithm in fact runs in $O(n^2 \sqrt{m})$ time. Our discussion in Section 7.7 presents a simplified proof of Cheriyan and Maheshwari approach. Ahuja and Orlin [1989] developed the excess scaling algorithm described in Section 7.9; this algorithm runs in $O(nm + n^2 \log U)$ time and obtains dramatic improvements over the FIFO and highest-label preflow-push algorithms without using sophisticated data structures. Ahuja, Orlin, and Tarjan [1989] further improved the excess scaling algorithm and obtained several algorithms: the best time bound of these algorithms is $O(nm \log(n \sqrt{\log U}/m + 2))$.

Cheriyan and Hagerup [1989] proposed a randomized algorithm for the maximum flow problem that has an expected running time of $O(nm)$ for all $m \geq n \log^2 n$. Alon [1990] developed a nonrandomized version of this algorithm and obtained a (deterministic) maximum flow algorithm that runs in (1) $O(nm)$ time for all $m = \Omega(n^{5/3} \log n)$, and (2) $O(nm \log n)$ time for all other values of n and m. Cheriyan, Hagerup, and Mehlhorn [1990] obtained an $O(n^3/\log n)$ algorithm for the maximum flow problem. Currently, the best available time bounds for solving the maximum flow problem are due to Alon [1990], Ahuja, Orlin, and Tarjan [1989], and Cheriyan, Hagerup, and Mehlhorn [1990].

Researchers have also investigated whether the worst-case bounds of the maximum flow algorithms are "tight" (i.e., whether algorithms achieve their worst-case bounds for some families of networks). Galil [1981] constructed a family of networks and showed that the algorithms of Edmonds and Karp [1972], Dinic [1970], Karzanov [1974], and a few other maximum flow algorithms achieve their worst-case bounds. Using this family of networks, it is possible to show that the shortest augmenting path algorithm also runs in $\Omega(n^2 m)$ time. Cheriyan and Maheshwari [1989] have shown that the generic preflow-push algorithm and its FIFO and the highest-label preflow-push implementations run in $\Omega(n^2 m)$, $\Omega(n^3)$, and $\Omega(n^2 \sqrt{m})$ times, respectively. Thus the worst-case time bounds of these algorithms are tight.

Several computational studies have assessed the empirical behavior of maximum flow algorithms. Among these, the studies by Imai [1983], Glover, Klingman, Mote, and Whitman [1984], Derigs and Meier [1989], and Ahuja, Kodialam, Mishra, and Orlin [1992] are noteworthy. These studies find that preflow-push algorithms are faster than augmenting path algorithms. Among the augmenting path algorithms, the shortest augmenting path algorithm is the fastest, and among the preflow-push algorithms, the performance of the highest-label preflow-push algorithm is the most attractive.

EXERCISES

7.1. Consider the network shown in Figure 7.20. The network depicts only those arcs with a positive capacity. Specify the residual network with respect to the current flow and compute exact distance labels in the residual network. Next change the arc flows (without changing the flow into the sink) so that the exact distance label of the source node (1) decreases by 1 unit; (2) increases by 1 unit.

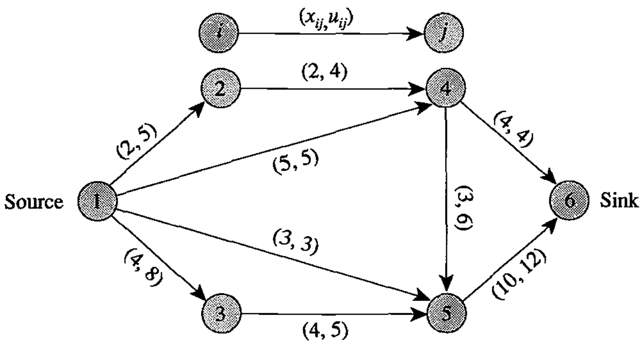


Figure 7.20 Example for Exercise 7.1.

7.2. Using the capacity scaling algorithm described in Section 7.3, find a maximum flow in the network given in Figure 7.21(b).

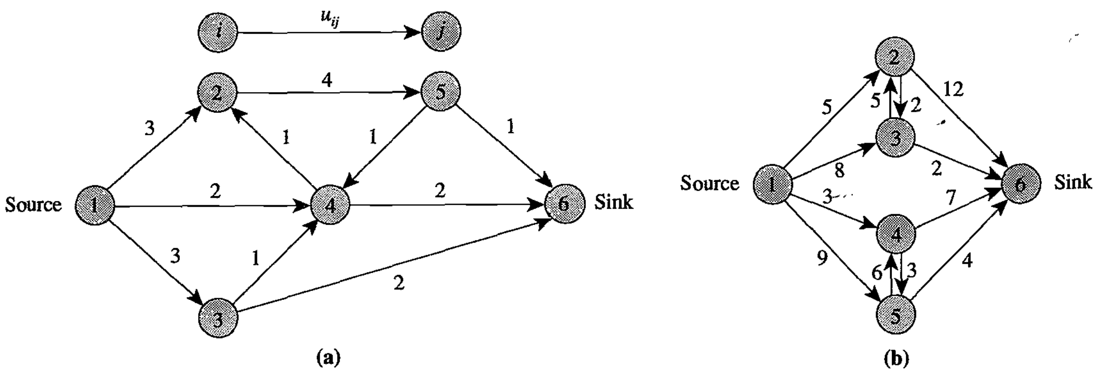


Figure 7.21 Examples for Exercises 7.2, 7.3, 7.5, and 7.6.

7.3. Using the shortest augmenting path algorithm, solve the maximum flow problem shown in Figure 7.21(a).

7.4. Solve the maximum flow problem shown in Figure 7.22 using the generic preflow-push algorithm. Incorporate the following rules to maintain uniformity of your computations: (1) Select an active node with the smallest index. [For example, if nodes 2 and 3 are active, select node 2.] (2) Examine the adjacency list of any node in the increasing order of the head node indices. [For example, if $A(1) = \{(1, 5), (1, 2), (1, 7)\}$, then examine arc (1, 2) first.] Show your computations on the residual networks encountered during the intermediate iterations of the algorithm.

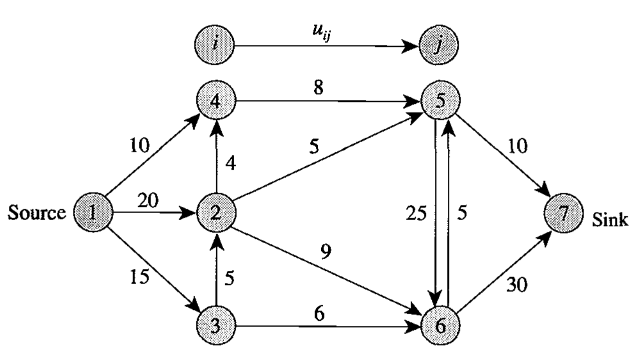


Figure 7.22 Example for Exercise 7.4.

7.5. Solve the maximum flow problem shown in Figure 7.21(a) using the FIFO preflow-push algorithm. Count the number of saturating and nonsaturating pushes and the number of relabel operations. Next, solve the same problem using the highest-label preflow-push algorithm. Compare the number of saturating pushes, nonsaturating pushes, and relabel operations with those of the FIFO preflow-push algorithm.

7.6. Using the excess scaling algorithm, determine a maximum flow in the network given in Figure 7.21(b).

7.7. **Most vital arcs.** We define a *most vital arc* of a network as an arc whose deletion causes the largest decrease in the maximum flow value. Either prove the following claims or show through counterexample that they are false.
- (a) A most vital arc is an arc with the maximum value of u_{ij}.
- (b) A most vital arc is an arc with the maximum value of x_{ij}.
- (c) A most vital arc is an arc with the maximum value of x_{ij} among arcs belonging to some minimum cut.
- (d) An arc that does not belong to some minimum cut cannot be a most vital arc.
- (e) A network might contain several most vital arcs.

7.8. **Least vital arcs.** A *least vital arc* in a network is an arc whose deletion causes the least decrease in the maximum flow value. Either prove the following claims or show that they are false.
- (a) Any arc with $x_{ij} = 0$ in any maximum flow is a least vital arc.
- (b) A least vital arc is an arc with the minimum value of x_{ij} in a maximum flow.
- (c) Any arc in a minimum cut cannot be a least vital arc.

7.9. Indicate which of the following claims are true or false. Justify your answer by giving a proof or by constructing a counterexample.
- (a) If the capacity of every arc in a network is a multiple of α, then in every maximum flow, each arc flow will be a multiple of α.
- (b) In a network G, if the capacity of every arc increases by α units, the maximum flow value will increase by a multiple of α.
- (c) Let v^* denote the maximum flow value of a given maximum flow problem. Let v' denote the flow into the sink node t at some stage of the preflow-push algorithm. Then $v^* - v' \leq \sum_{i \text{ is active}} e(i)$.
- (d) By the flow decomposition theory, some sequence of at most $m + n$ augmentations would always convert any preflow into a maximum flow.
- (e) In the excess scaling algorithm, $e_{\max} = \max\{e(i): i \text{ is active}\}$ is a nonincreasing function of the number of push/relabel steps.
- (f) The capacity of the augmenting paths generated by the maximum capacity augmenting path algorithm is nonincreasing.
- (g) If each distance label $d(i)$ is a lower bound on the length of a shortest path from node i to node t in the residual network, the distance labels are valid.

7.10. Suppose that the capacity of every arc in a network is a multiple of α and is in the range $[0, \alpha K]$ for some integer K. Does this information improve the worst-case complexity of the labeling algorithm, FIFO preflow-push algorithm, and the excess scaling algorithm?

7.11. **Converting a maximum preflow to a maximum flow.** We define a *maximum preflow* x° as a preflow with the maximum possible flow into the sink.

(a) Show that for a given maximum preflow x^0, some maximum flow x^* with the same flow value as x^0, satisfies the condition that $x^*_{ij} \le x^0_{ij}$ for all arcs $(i, j) \in A$. (*Hint*: Use flow decomposition.)

(b) Suggest a labeling algorithm that converts a maximum preflow into a maximum flow in at most $n + m$ augmentations.

(c) Suggest a variant of the shortest augmenting path algorithm that would convert a maximum preflow into a maximum flow in $O(nm)$ time. (*Hint*: Define distance labels from the source node and show that the algorithm will create at most m arc saturations.)

(d) Suggest a variant of the highest-label preflow-push algorithm that would convert a maximum preflow into a maximum flow. Show that the running time of this algorithm is $O(nm)$. (*Hint*: Use the fact that we can delete an arc with zero flow from the network.)

7.12. (a) An arc is *upward critical* if increasing the capacity of this arc increases the maximum flow value. Does every network have an upward critical arc? Describe an algorithm for identifying all upward critical arcs in a network. The worst-case complexity of your algorithm should be substantially better than that of solving m maximum flow problems.

(b) An arc is *downward critical* if decreasing the capacity of this arc decreases the maximum flow value. Is the set of upward critical arcs the same as the set of downward critical arcs? If not, describe an algorithm for identifying all downward critical arcs; analyze your algorithm's worst-case complexity.

7.13. Show that in the shortest augmenting path algorithm or in the preflow-push algorithm, if an arc (i, j) is inadmissible at some stage, it remains inadmissible until the algorithm relabels node i.

7.14. Apply the generic preflow-push algorithm to the maximum flow problem shown in Figure 7.23. Always examine a node with the smallest distance label and break ties in favor of a node with the smallest node number. How many saturating and nonsaturating pushes does the algorithm perform?

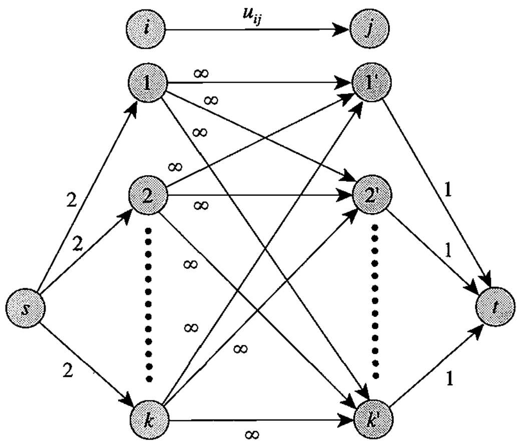


Figure 7.23 Example for Exercise 7.14.

7.15. Apply the FIFO preflow-push algorithm to the network shown in Figure 7.24. Determine the number of pushes as a function of the parameters W and L (correct within a constant factor). For a given value of n, what values of W and L produce the largest number of pushes?

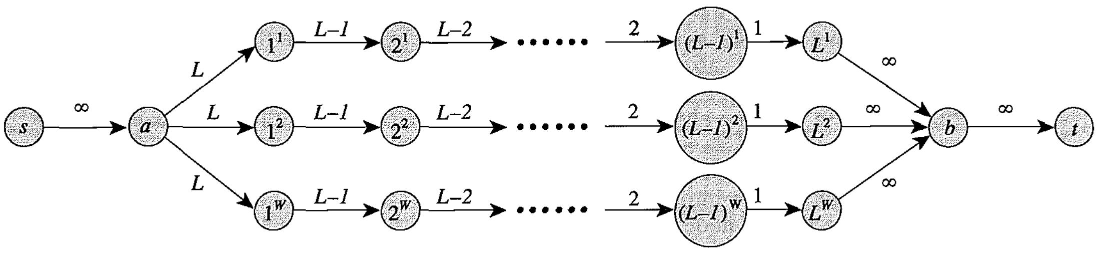


Figure 7.24 Example for Exercise 7.15.

7.16. Apply the highest-label preflow-push algorithm on the network shown in Figure 7.24. Determine the number of pushes as a function of the parameters W and L (correct within a constant factor). For a given n, what values of W and L produce the largest number of pushes?

7.17. Describe a more general version of the capacity scaling algorithm discussed in Section 7.3, one that scales Δ at each scaling phase by a factor of some integer number $\beta \geq 2$. Initially, $\Delta = \beta^{\lceil \log_\beta U \rceil}$ and each scaling phase reduces Δ by a factor of β. Analyze the worst-case complexity of this algorithm and determine the optimal value of β.

7.18. **Partially capacitated networks** (Ahuja and Orlin [1991]). Suppose that we wish to speed up the capacity scaling algorithm discussed in Exercise 7.17 for networks with some, but not all, arcs capacitated. Suppose that the network G has p arcs with finite capacities and $\beta = \max\{2, \lceil m/p \rceil\}$. Consider a version of the capacity scaling algorithm that scales Δ by a factor of β in each scaling phase. Show that the algorithm would perform at most $2m$ augmentations per scaling phase. [*Hint*: At the end of the Δ-scaling phase, the s–t cut in $G(\Delta)$ contains only arcs with finite capacities.] Conclude that the capacity scaling algorithm would solve the maximum flow problem in $O(m^2 \log_\beta U)$ time. Finally, show that this algorithm would run in $O(m^2)$ time if $U = O(n^k)$ for some k and $m = O(n^{1+\epsilon})$ for some $\epsilon > 0$.

7.19. **Bit-scaling algorithm** (Gabow [1985]). Let $K = \lceil \log U \rceil$. In the bit-scaling algorithm for the maximum flow problem works, we represent each arc capacity as a K-bit binary number, adding leading zeros if necessary to make each capacity K bits long. The problem P_k considers the capacity of each arc as the k leading bits. Let x_k^* denote a maximum flow and let v_k^* denote the maximum flow value in the problem P_k. The algorithm solves a sequence of problems $P_1, P_2, P_3, \ldots, P_K$, using $2x_{k-1}^*$ as a starting solution for the problem P_k.

(a) Show that $2x_{k-1}^*$ is feasible for P_k and that $v_k^* - 2v_{k-1}^* \leq m$.

(b) Show that the shortest augmenting path algorithm for solving problem P_k, starting with $2x_{k-1}^*$ as the initial solution, requires $O(nm)$ time. Conclude that the bit-scaling algorithm solves the maximum flow problem in $O(nm \log U)$ time.

7.20. Using the potential function $\Phi = \max\{d(i) : i \text{ is active}\}$, show that the highest-label preflow-push algorithm performs $O(n^3)$ nonsaturating pushes.

7.21. The *wave algorithm*, which is a hybrid version of the highest-label and FIFO preflow-push algorithms, performs passes over active nodes. In each pass it examines *all* the active nodes in nonincreasing order of the distance labels. While examining a node, it pushes flow from a node until either its excess becomes zero or the node is relabeled. If during a pass, the algorithm relabels no node, it terminates; otherwise, in the next pass, it again examines active nodes in nonincreasing order of their new distance labels.

Discuss the similarities and differences between the wave algorithm with the highest label and FIFO preflow-push algorithms. Show that the wave algorithm runs in $O(n^3)$ time.

7.22. Several rules listed below are possible options for selecting an active node to perform the push/relabel operation in the preflow-push algorithm. Describe the data structure and the implementation details for each rule. Obtain the tightest possible bound on the numbers of pushes performed by the algorithm and the resulting running time of the algorithm.
(a) Select an active node with the minimum distance label.
(b) Select an active node with the largest amount of excess.
(c) Select an active node whose excess is at least 50 percent of the maximum excess at any node.
(d) Select the active node that the algorithm had selected most recently.
(e) Select the active node that the algorithm had selected least recently.
(f) Select an active node randomly. (Assume that for any integer k, you can in $O(1)$ steps generate a random integer uniformly in the range $[1, k]$.)

7.23. In the excess scaling algorithm, suppose we require that each nonsaturating push pushes exactly $\Delta/2$ units of flow. Show how to modify the push/relabel step to meet this requirement. Does this modification change the worst-case running time of the algorithm?

7.24. In our development in this chapter we showed that the excess scaling algorithm performs $O(n^2 \log U^*)$ nonsaturating pushes if U^* is set equal to the largest arc capacity among the arcs emanating from the source node. However, we can often select smaller values of U^* and still show that the number of nonsaturating pushes is $O(n^2 \log U^*)$. Prove that we can also use the following values of U^* without affecting the worst-case complexity of the algorithm: (1) $U^* = \sum_{(s,j)\in A(s)} u_{sj}/\,|A(s)|$; (2) $U^* = v^{\text{ub}}/\,|A(s)|$ for any upper bound v^{ub} on the maximum flow value. (*Hint*: In the first scaling phase, set $\Delta = 2^{\lceil \log U^* \rceil}$ and forbid nodes except those adjacent to the source from having excess more than Δ.)

7.25. The excess scaling algorithm described in Section 7.9 scales excesses by a factor of 2. It starts with the value of the excess dominator Δ equal to the smallest power of 2 that is greater than or equal to U; in every scaling phase, it reduces Δ by a factor of 2. An alternative is to scale the excesses by a factor of some integer number $\beta \geq 2$. This algorithm would run as follows. It would start with $\Delta = \beta^{\lceil \log_\beta U \rceil}$; it would then reduce Δ by a factor of β in every scaling phase. In the Δ-scaling phase, we refer to a node with an excess of at least Δ/β as a node with a *large* excess. The algorithm pushes flow from a node with a large excess and among these nodes it chooses the node with the smallest distance label. The algorithm also ensures that no excess exceeds Δ. Determine the number of nonsaturating pushes performed by the algorithm as a function of n, β, and U. For what value of β would the algorithm perform the least number of nonsaturating pushes?

7.26. For any pair $[i, j] \in N \times N$, we define $\alpha[i, j]$ in the following manner: (1) if $(i, j) \in A$, then $\alpha[i, j]$ is the increase in the maximum flow value obtained by setting $u_{ij} = \infty$; and (2) if $(i, j) \notin A$, then $\alpha[i, j]$ is the increase in the maximum flow value obtained by introducing an infinite capacity arc (i, j) in the network.
(a) Show that $\alpha[i, j] \leq \alpha[s, j]$ and $\alpha[i, j] \leq \alpha[i, t]$.
(b) Show that $\alpha[i, j] = \min\{\alpha[s, j], \alpha[i, t]\}$.
(c) Show that we can compute $\alpha[i, j]$ for all node pairs by solving $O(n)$ maximum flow problems.

7.27. **Minimum cut with the fewest number of arcs.** Suppose that we wish to identify from among all minimum cuts, a minimum cut containing the least number of arcs. Show that if we replace u_{ij} by $u'_{ij} = mu_{ij} + 1$, the minimum cut with respect to the capacities u'_{ij} is a minimum cut with respect to the capacities u_{ij} containing the fewest number of arcs.

7.28. Parametric network feasibility problem. In a capacitated network G with arc capacities u_{ij}, suppose that the supply/demands of nodes are linear functions of time τ. Let each $b(i) = b^0(i) + \tau b^*(i)$ and suppose that $\sum_{i \in N} b^0(i) = 0$ and $\sum_{i \in N} b^*(i) = 0$. The network is currently (i.e., at time $\tau = 0$) able to fulfill the demands by the existing supplies but might not be able to do so at some point in future. You want to determine the largest integral value of τ up to which the network will admit a feasible flow. How would you solve this problem?

7.29. Source parametric maximum flow problem (Gallo, Grigoriadis, and Tarjan [1989]). In the *source parametric maximum flow problem*, the capacity of every source arc (s, j) is a nondecreasing linear function of a parameter λ (i.e., $u_{sj} = u^{\circ}_{sj} + \lambda u^*_{sj}$ for some constant $u^*_{sj} \geq 0$); the capacity of every other arc is fixed, and we wish to determine a maximum flow for p values $0 = \lambda_1, \lambda_2, \ldots, \lambda_p$ of the parameter λ. Assume that $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_p$ and $p \leq n$. As an application of the source-parametric maximum flow problem, consider the following variation of Application 6.5. Suppose that processor 1 is a shared multiprogrammed system and processor 2 is a graphic processor dedicated to a single user. Suppose further that we can accurately determine the times required for processing modules on processor 2, but the times for processing modules on processor 1 are affected by a general work load on the processor. As the work load on processor 1 changes, the optimal distribution of modules between processor 1 and 2 changes. The source–parametric maximum flow problem determines these distributions for different work loads on processor 1.

Let MF(λ) denote the maximum flow problem for a specific value of λ. Let $v(\lambda)$ denote the maximum flow value of MF(λ) and let $[S(\lambda), \bar{S}(\lambda)]$ denote an associated minimum cut. Clearly, the zero flow is optimal for MF(λ_1). Given an optimal flow $x(\lambda_k)$ of MF(λ_k), we solve MF(λ_{k+1}) as follows: With $x(\lambda_k)$ as the starting flow and the corresponding distance labels as the initial distance labels, we perform a preprocess step by sending additional flow along the source arcs so that they all become saturated. Then we apply the FIFO preflow-push algorithm until the network contains no more active nodes. We repeat this process until we have solved MF(λ_p).

(a) Show that the ending distance labels of MF(λ_k) are valid distances for MF(λ_{k+1}) in the residual network $G(x(\lambda_k))$ after the preprocess step.

(b) Use the result in part (a) to show that overall [i.e., in solving all the problems MF(λ_1), MF(λ_2), . . . , MF(λ_p)], the algorithm performs $O(n^2)$ relabels, $O(nm)$ saturating pushes, and $O(n^3)$ nonsaturating pushes. Conclude that the FIFO preflow-push algorithm solves the source parametric maximum flow problem in $O(n^3)$ time, which is the same time required to solve a single maximum flow problem.

(c) Show that $v(\lambda_1) \leq v(\lambda_2) \leq \cdots \leq v(\lambda_p)$ and some associated minimum cuts satisfy the nesting condition $S_1 \subseteq S_2 \subseteq \cdots \subseteq S_p$.

7.30. Source–sink parametric maximum flow problem. In the source–sink parametric maximum flow problem, the capacity of every source arc is a *nondecreasing* linear function of a parameter λ and capacity of every sink arc is a *nonincreasing* linear function of λ, and we want to determine a maximum flow for several values of parameter $\lambda_1, \lambda_2, \ldots, \lambda_p$, for $p \leq n$, that satisfy the condition $0 = \lambda_1 < \lambda_2 < \cdots < \lambda_p$. Show how to solve this problem in a total of $O(n^3)$ time. (*Hint*: The algorithm is same as the one considered in Exercise 7.29 except that in the preprocess step if some sink arc has flow greater than its new capacity, we decrease the flow.)

7.31. Ryser's theorem. Let Q be a $p \times p$ matrix consisting of 0–1 elements. Let α denote the vector of row sums of Q and β denote the vector of column sums. Suppose that the rows and columns are ordered so that $\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_p$, and $\beta_1 \geq \beta_2 \geq \cdots \geq \beta_p$.

(a) Show that the vectors α and β must satisfy the following conditions: (1) $\sum_{i=1}^p \alpha_i = \sum_{i=1}^p \beta_i$ and (2) $\sum_{i=1}^k \min(\alpha_i, k) \leq \sum_{i=1}^k \beta_i$, for all $k = 1, \ldots, p$. [*Hint*: $\min(\alpha_i, k)$ is an upper bound on the sum of the first k components of row i.]

(b) Given the nonnegative integer vector α and β, show how to formulate the problem of determining whether some 0–1 matrix Q has a row sum vector α and a column sum vector β as a maximum flow problem. Use the max-flow min-cut theorem to show that the conditions stated in part (a) are sufficient for the existence of such a matrix Q.

8

MAXIMUM FLOWS: ADDITIONAL TOPICS

This was the most unkindest cut of all.
—Shakespeare in Julius Caeser Act III

Chapter Outline

8.1 INTRODUCTION

In all scientific disciplines, researchers are always making trade-offs between the generality and the specificity of their results. Network flows embodies these considerations. In studying minimum cost flow problems, we could consider optimization models with varying degrees of generality: for example, in increasing order of specialization, (1) general constrained optimization problems, (2) linear programs, (3) network flows, (4) particular network flow models (e.g., shortest path and maximum flow problems), and (5) the same models defined on problems with specific topologies and/or cost structures. The trade-offs in choosing where to study across the hierarchy of possible models is apparent. As models become broader, so does the range of their applications. As the models become more narrow, available results often become refined and more powerful. For example, as shown by our discussion in previous chapters, algorithms for shortest path and maximum flow problems have very attractive worst-case and empirical behavior. In particular, the computational complexity of these algorithms grows rather slowly in the number of underlying constraints (i.e., nodes) and decision variables (arcs). For more general linear programs, or even for more general minimum cost flow problems, the best algorithms are not nearly as good.

In considering what class of problems to study, we typically prefer models that are generic enough to be rich, both in applications and in theory. As evidenced by the coverage in this book, network flows is a topic that meets this criterion. Yet, through further specialization, we can develop a number of more refined results. Our study of shortest paths and maximum flow problems in the last four chapters has illustrated this fact. Even within these more particular problem classes, we have seen the effect of further specialization, which has led to us to discover more efficient shortest path algorithms for models with nonnegative costs and for models defined on acyclic graphs. In this chapter we carry out a similar program for maximum flow problems. We consider maximum flow problems with both (1) specialized data, that is, networks with unit capacity arcs, and (2) specialized topologies, namely, bipartite and planar networks.

For general maximum flow problems, the labeling algorithm requires $O(nmU)$ computations and the shortest augmenting path algorithm requires $O(n^2m)$ computations. When applied to unit capacity networks, these algorithms are guaranteed to perform even better. Both require $O(nm)$ computations. We obtain this improvement simply because of the special nature of unit capacity networks. By designing specialized algorithms, however, we can improve even further on these results. Combining features of both the labeling algorithm and the shortest augmenting path algorithm, the unit capacity maximum flow algorithm that we consider in this chapter requires only $O(\min\{n^{2/3}m, m^{3/2}\})$ computations.

Network connectivity is an important application context for the unit capacity maximum flow problem. The arc connectivity between any two nodes of a network is the maximum number of arc-disjoint paths that connect these nodes; the arc connectivity of the network as a whole is the minimum arc connectivity between any pair of nodes. To determine this important reliability measure of a network, we could solve a unit capacity maximum flow problem between every pair of nodes, thus requiring $O(\min\{n^{2/3}m, m^{3/2}\})$ computations. As we will see in this chapter, by exploiting the special structure of the arc connectivity problem, we can reduce this complexity bound considerably—to $O(nm)$.

For networks with specialized bipartite and planar topologies, we can also obtain more efficient algorithms. Recall that bipartite networks are composed of two node sets, N_1 with n_1 nodes and N_2 with n_2 nodes. Assume that $n_1 \leq n_2$. For these problems we develop a specialization of the generic preflow-push algorithm that requires $O(n_1^2\, m)$ instead of $O((n_1 + n_2)^2\, m)$ time. Whenever the bipartite network is unbalanced in the sense that $n_1 \ll (n_1 + n_2) = n$, the new implementation has a much better complexity than the general preflow-push algorithm. Planar networks are those that we can draw on the plane so that no two arcs intersect each other. For this class of networks, we develop a specialized maximum flow algorithm that requires only $O(n \log n)$ computations.

In this chapter we also consider two other additional topics: a dynamic tree implementation and the all-pairs minimum value cut problem. Dynamic trees is a special type of data structure that permits us to implicitly send flow on paths of length n in $O(\log n)$ steps on average. By doing so we are able to reduce the computational requirement of the shortest augmenting path algorithm for maximum flows from $O(n^2m)$ to $O(nm \log n)$.

In some application contexts, we need to find the maximum flow between every pair of nodes in a network. The max-flow min-cut theorem shows that this problem is equivalent to finding the minimum cut separating all pairs of nodes. The most naive way to solve this problem would be to solve the maximum flow problem $n(n-1)$ times, once between every pair of nodes. Can we do better? In Section 8.7 we show that how to exploit the relationship of the cut problems between various node pairs to reduce the computational complexity of the all-pairs minimum value cut problem considerably in undirected networks. This algorithm requires solving only $(n - 1)$ minimum cut problems in undirected networks. Moreover, the techniques used in this development extend to a broader class of problems: they permit us to solve the all-pairs minimum cut problem for situations when the value of a cut might be different than the sum of the arc capacities across the cut.

The algorithms we examine in this chapter demonstrate the advantage of exploiting special structures to improve on the design of algorithms. This theme not only resurfaces on several other occasions in this book, but also is an important thread throughout the entire field of large-scale optimization. Indeed, we might view the field of large-scale optimization, and the field of network flows for that matter, as the study of theory and algorithms for exploiting special problem structure. In this sense this chapter is, in its orientation and overall approach, a microcosm of this entire book and of much of the field of optimization itself.

8.2 FLOWS IN UNIT CAPACITY NETWORKS

Certain combinatorial problems are naturally formulated as zero–one optimization models. When viewed as flow problems, these models yield networks whose arc capacities are all 1. We will refer to these networks as *unit capacity networks*. Frequently, it is possible to solve flow problems on these networks more efficiently than those defined on general networks. In this section we describe an efficient algorithm for solving the maximum flow problem on unit capacity networks. We subsequently refer to this algorithm as the *unit capacity maximum flow algorithm*.

In a unit capacity network, the maximum flow value is at most n, since the capacity of the s–t cut $[\{s\}, S - \{s\}]$ is at most n. The labeling algorithm therefore determines a maximum flow within n augmentations and requires $O(nm)$ effort. The shortest augmenting path algorithm also solves this problem in $O(nm)$ time since its bottleneck operation, which is the augmentation step, requires $O(nm)$ time instead of $O(n^2m)$ time. The unit capacity maximum flow algorithm that we describe is a hybrid version of these two algorithms. This unit capacity maximum flow algorithm is noteworthy because by combining features of both algorithms, it requires only $O(\min\{n^{2/3}m, m^{3/2}\})$ time, which is consistently better than the $O(nm)$ bound of either algorithm by itself.

The unit capacity maximum flow algorithm is a two-phase algorithm. In the first phase it applies the shortest augmenting path algorithm, although not until completion: rather, this phase terminates whenever the distance label of the source node satisfies the condition $d(s) \geq d^* = \min\{\lceil 2n^{2/3} \rceil, \lceil m^{1/2} \rceil\}$. Although the algorithm might terminate with a nonoptimal solution, the solution is probably nearly-optimal (its value is within d^* of the optimal flow value). In its second phase, the algorithm applies the labeling algorithm to convert this near-optimal flow into a maximum flow. As we will see, this two-phase approach works well for unit capacity networks because the shortest augmenting path algorithm obtains a near-optimal flow quickly (when augmenting paths are "short") but then takes a long time to convert this solution into a maximum flow (when augmenting paths become "long"). It so happens that the labeling algorithm converts this near-optimal flow into a maximum flow far more quickly than the shortest augmenting path algorithm.

Let us examine the behavior of the shortest augmenting path algorithm for $d^* = \min\{\lceil 2n^{2/3} \rceil, \lceil m^{1/2} \rceil\}$. Suppose the algorithm terminates with a flow vector x' with a flow value equal to v'. What can we say about $v^* - v'$? (Recall that v^* denotes the maximum flow value.) We shall answer this question in two parts: (1) when $d^* = \lceil 2n^{2/3} \rceil$, and (2) when $d^* = \lceil m^{1/2} \rceil$.

Suppose that $d^* = \lceil 2n^{2/3} \rceil$. For each $k = 0, 1, 2, \ldots, d^*$, let V_k denote the set of nodes with a distance label equal to k [i.e., $V_k = \{i \in N: d(i) = k\}$]. We refer to V_k as the set of nodes in the kth *layer* of the residual network. Consider the situation when each of the sets $V_1, V_2, \ldots, V_{d^*}$ is nonempty. It is possible to show that each arc (i, j) in the residual network $G(x')$ connects a node in the kth layer to a node in the $(k + 1)$th layer for some k, for otherwise $d(i) > d(j) + 1$, which contradicts the distance label validity conditions (7.2). Therefore, for each $k = 1, 2, \ldots, d^*$, the set of arcs joining the node sets V_k to V_{k-1} form an s–t cut in the residual network. In case one of the sets, say V_k, is empty, our discussion in Section 7.4 implies that the cut $[S, \bar{S}]$ defined by $S = V_{k+1} \cup V_{k+2} \cup \cdots \cup V_{d^*}$ is a minimum cut.

Note that $|V_1| + |V_2| + \cdots + |V_{d^*}| \le n - 1$, because the sink node does not belong to any of these sets. We claim that the residual network contains at least two consecutive layers V_k and V_{k-1}, each with at most $n^{1/3}$ nodes. For if not, every alternate layer (say, $V_1, V_3, V_5, \ldots$) must contain more than $n^{1/3}$ nodes and the total number of nodes in these layers would be strictly greater than $n^{1/3}\, d^*/2 \ge n$, leading to a contradiction. Consequently, $|V_k| \le n^{1/3}$ and $|V_{k-1}| \le n^{1/3}$ for some of the two layers V_k and V_{k-1}. The residual capacity of the s–t cut defined by the arcs connecting V_k to V_{k-1} is at most $|V_k|\,|V_{k-1}| \le n^{2/3}$ (since at most one arc of unit residual capacity joins any pair of nodes). Therefore, by Property 6.2, $v^* - v' \le n^{2/3} \le d^*$.

Next consider the situation when $d^* = \lceil m^{1/2} \rceil$. The layers of nodes $V_1, V_2, \ldots, V_{d^*}$ define d^* s–t cuts in the residual network and these cuts are arc disjoint in $G(x)$. The sum of the residual capacities of these cuts is at most m since each arc contributes at most one to the residual capacity of any such cut. Thus some s–t cut must have residual capacity at most $\lceil m^{1/2} \rceil$. This conclusion proves that $v^* - v' \le \lceil m^{1/2} \rceil = d^*$.

In both cases, whenever $d^* = \lceil 2n^{2/3} \rceil$ or $d^* = \lceil m^{1/2} \rceil$, we find that the first phase obtains a flow whose value differs from the maximum flow value by at most d^* units. The second phase converts this flow into a maximum flow in $O(d^*m)$ time since each augmentation requires $O(m)$ time and carries a unit flow. We now show that the first phase also requires $O(d^*m)$ time.

In the first phase, whenever the distance label of a node k exceeds d^*, this node never occurs as an intermediate node in any subsequent augmenting path since $d(k) < d(s) < d^*$. So the algorithm relabels any node at most d^* times. This observation gives a bound of $O(d^*n)$ on the number of retreat operations and a bound of $O(d^*m)$ on the time to perform the retreat operations. Consider next the augmentation time. Since each arc capacity is 1, flow augmentation over an arc immediately saturates that arc. During two consecutive saturations of any arc (i, j), the distance labels of both the nodes i and j must increase by at least 2 units. Thus the algorithm can saturate any arc at most $\lfloor d^*/2 \rfloor$ times, giving an $O(d^*m)$ bound on the total time needed for flow augmentations. The total number of advance operations is bounded by the augmentation time plus the number of retreat operations and is again $O(d^*m)$. We have established the following result.

Theorem 8.1. *The unit capacity maximum flow algorithm solves a maximum flow problem on unit capacity networks in $O(min\{n^{2/3}m, m^{3/2}\})$ time.* ◆

The justification of this two-phase procedure should now be clear. If $d^* = \lceil 2n^{2/3} \rceil \le m^{1/2}$, the preceding discussion shows that the shortest augmenting path algorithm requires $O(n^{2/3}m)$ computations to obtain a flow within $n^{2/3}$ of the optimal. If we allow this algorithm to run until it achieves optimality [i.e., until $d(s) \ge n$], the algorithm could require an additional $O((n - n^{2/3})m)$ time to convert this flow into an optimal flow. For $n = 1000$, these observations imply that if the algorithm achieves these bounds, it requires 10 percent of the time to send 90 percent of the maximum flow and the remaining 90 percent of the time to send 10 percent of the maximum flow. (Empirical investigations have observed a similar behavior in practice as well.) On the other hand, the use of labeling algorithm in the second phase establishes a maximum flow in $O(n^{2/3}m)$ time and substantially speeds up the overall performance of the algorithm.

Another special case of unit capacity networks, called *unit capacity simple networks*, also arises in practice and is of interest to researchers. For this class of unit capacity networks, every node in the network, except the source and sink nodes, has at most one incoming arc or at most one outgoing arc. The unit capacity maximum flow algorithm runs even faster for this class of networks. We achieve this improvement by setting $d^* = \lceil n^{1/2} \rceil$ in the algorithm.

Theorem 8.2. *The unit capacity maximum flow algorithm establishes a maximum flow in unit capacity simple networks in $O(n^{1/2}m)$ time.*

Proof. Consider the layers of nodes $V_1, V_2, \ldots, V_{d^*}$ at the end of the first phase. Note first that $d(s) > d^*$ since otherwise we could find yet another augmentation in Phase 1. Suppose that layer V_h contains the smallest number of nodes. Then $|V_h| \le n^{1/2}$, since otherwise the number of nodes in all layers would be strictly greater than n. Let N' be the nodes in N with at most one outgoing arc. We define a cut $[S, N\text{-}S]$ as follows: $S = \{j:d(j) \ge h\} \cup \{j:d(j) = h \text{ and } j \in N'\}$. Since $d(s) > d^* \ge h$ and $d(t) = 0$, $[S, N\text{-}S]$ is an s–t cut. Each arc with residual capacity in the cut $[S, N\text{-}S]$ is either directed into a node in $V_h \cap (N\text{-}N')$ or else it is directed from a node in $V_h \cap N'$. Therefore, the residual capacity of the cut Q is at most $|V_h| \le n^{1/2}$. Consequently, at the termination of the first phase, the flow value differs from the maximum flow value by at most $n^{1/2}$ units. Using arguments similar to those we have just used, we can now easily show that the algorithm would run in $O(n^{1/2}m)$ time. ◆

The proof of the Theorem 8.2 relies on the fact that only 1 unit of flow can pass through each node in the network (except the source and sink nodes). If we satisfy this condition but allow some arc capacities to be larger than 1, the unit capacity maximum flow algorithm would still require only $O(n^{1/2}m)$ time. Networks with this structure do arise on occasion; for example, we encountered this type of network when we computed the maximum number of node-disjoint paths from the source node to the sink node in the proof of Theorem 6.8. Theorem 8.2 has another by-product: It permits us to solve the maximum bipartite matching problem in $O(n^{1/2}m)$ time since we can formulate this problem as a maximum flow problem on a unit capacity simple network. We study this transformation in Section 12.3.

8.3 FLOWS IN BIPARTITE NETWORKS

A *bipartite network* is a network $G = (N, A)$ with a node set N partitioned into two subsets N_1 and N_2 so that for every arc $(i, j) \in A$, either (1) $i \in N_1$ and $j \in N_2$, or (2) $i \in N_2$ and $j \in N_1$. We often represent a bipartite network using the notation $G = (N_1 \cup N_2, A)$. Let $n_1 = |N_1|$ and $n_2 = |N_2|$. Figure 8.1 gives an example of a bipartite network; in this case, we can let $N_1 = \{1, 2, 3, 9\}$ and $N_2 = \{4, 5, 6, 7, 8\}$.

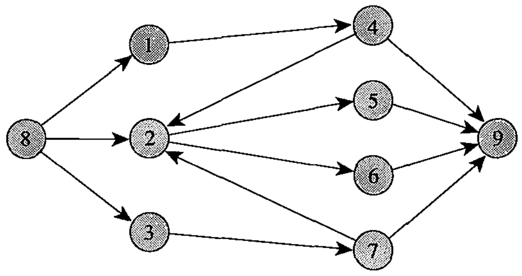


Figure 8.1 Bipartite network.

In this section we describe a specialization of the preflow-push algorithms that we considered in Chapter 7, but now adapt it to solve maximum flow problems on bipartite networks. The worst-case behavior of these special-purpose algorithms is similar to those of the original algorithms if the node sets N_1 and N_2 are of comparable size; the new algorithms are considerably faster than the original algorithms, however, whenever one of the sets N_1 or N_2 is substantially larger than the other. Without any loss of generality, we assume that $n_1 \leq n_2$. We also assume that the source node belongs to N_2. [If the source node s belonged to N_1, then we could create a new source node $s' \in N_2$, and we could add an arc (s', s) with capacity M for sufficiently large M.] As one example of the type of results we will obtain, we show that the specialization of the generic preflow-push algorithm solves a maximum flow problem on bipartite networks in $O(n_1^2 m)$ time. If $n_1 \ll (n_1 + n_2) = n$, the new implementation is considerably faster than the original algorithm.

In this section we examine only the generic preflow-push algorithm for bipartite networks; we refer to this algorithm as the *bipartite preflow-push algorithm.* The ideas we consider also apply in a straightforward manner to the FIFO, highest-label preflow-push and excess-scaling algorithms and yield algorithms with improved worst-case complexity. We consider these improvements in the exercises.

We first show that a slightly modified version of the generic preflow-push algorithm requires less than $O(n^2m)$ time to solve problems defined on bipartite networks. To establish this result, we change the preprocess operation by setting $d(s) = 2n_1 + 1$ instead of $d(s) = n$. The modification stems from the observation that any path in the residual network can have at most $2n_1$ arcs since every alternate node in the path must be in N_1 (because the residual network is also bipartite) and no path can repeat a node in N_1. Therefore, if we set $d(s) = 2n_1 + 1$, the residual network will never contain a directed path from node s to node t, and the algorithm will terminate with a maximum flow.

Lemma 8.3. *For each node $i \in N$, $d(i) < 4n_1 + 1$.*

Proof. The proof is similar to that of Lemma 7.12. ◆

The following result is a direct consequence of this lemma.

Lemma 8.4.

(a) *Each distance label increases at most $O(n_1)$ times. Consequently, the total number of relabel operations is $O(n_1(n_1 + n_2)) = O(n_1 n_2)$.*

(b) *The number of saturating pushes is $O(n_1 m)$.*

Proof. The proofs are similar to those of Lemmas 7.13 and 7.14. ◆

It is possible to show that the results of Lemma 8.4 yield a bound of $O((n_1 m)n)$ on the number of nonsaturating pushes, as well as on the complexity of the generic preflow-push algorithm. Instead of considering the details of this approach, we next develop a modification of the generic algorithm that runs in $O(n_1^2 m)$ time.

This modification builds on the following idea. To bound the nonsaturating pushes of any preflow-push algorithm, we typically use a potential function defined in terms of all the active nodes in the network. If every node in the network can be active, the algorithm will perform a certain number of nonsaturating pushes. However, if we permit only the nodes in N_1 to be active, because $n_1 \leq n$ we can obtain a tighter bound on the number of nonsaturating pushes. Fortunately, the special structure of bipartite networks permits us to devise an algorithm that always manages to keep the nodes in N_2 inactive. We accomplish this objective by starting with a solution whose only active nodes are in N_1, and by performing pushes of length 2; that is, we push flow over two consecutive admissible arcs so that any excess always returns to a node in N_1 and no node in N_2 ever becomes active.

Consider the residual network of a bipartite network given in Figure 8.2(a), with node excesses and distance labels displayed next to the nodes, and residual capacities displayed next to the arcs. The bipartite preflow-push algorithm first pushes flow from node 1 because it is the only active node in the network. The algorithm then identifies an admissible arc emanating from node 1. Suppose that it selects the arc (1, 3). Since we want to find a path of length 2, we now look for an admissible arc emanating from node 3. The arc (3, 2) is one such arc. We perform a push on this path, pushing $\delta = \min\{e(1), r_{13}, r_{32}\} = \min\{6, 5, 4\} = 4$ units. This push saturates arc (3, 2), and completes one iteration of the algorithm. Figure 8.2(b) gives the solution at this point.

In the second iteration, suppose that the algorithm again selects node 1 as an active node and arc (1, 3) as an admissible arc emanating from this node. We would also like to find an admissible arc emanating from node 3, but the network has none. So we relabel node 3. As a result of this relabel operation, arc (1, 3) becomes in-

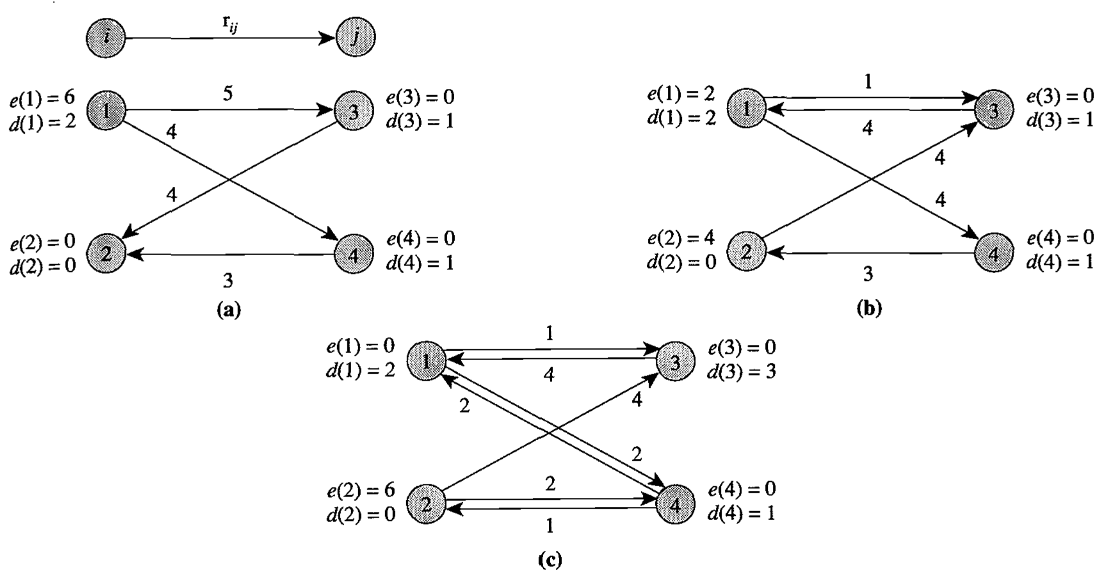


Figure 8.2 Illustrating the bipartite preflow-flow algorithm.

admissible. This operation completes the second iteration and Figure 8.2(b) gives the solution at this stage except that $d(3)$ is 3 instead of 1.

Suppose that the algorithm again selects node 1 as an active node in the third iteration. Then it selects the two consecutive admissible arcs (1, 4) and (4, 2), and pushes $\delta = \min\{e(1), r_{14}, r_{42}\} = \min\{2, 4, 3\} = 2$ units of flow over these arcs. This push is nonsaturating and eliminates the excess at node 1. Figure 8.2(c) depicts the solution at this stage.

As we have illustrated in this numerical example, the bipartite preflow-push algorithm is a simple generalization of the generic preflow-push algorithm. The bipartite algorithm is the same as the generic algorithm given in Figure 7.12 except that we replace the procedure push/relabel(i) by the procedure given in Figure 8.3.

```
procedure bipartite push/relabel(i);
begin
    if the residual network contains an admissible arc $(i, j)$ then
        if the residual network contains an admissible arc $(j, k)$ then
            push $\delta = \min\{e(i), r_{ij}, r_{jk}\}$ units of flow over the path $i$–$j$–$k$
        else replace $d(j)$ by $\min\{d(k) + 1 : (j, k) \in A(j)$ and $r_{jk} > 0\}$
    else replace $d(i)$ by $\min\{d(j) + 1 : (i, j) \in A(i)$ and $r_{ij} > 0\}$;
end;
```

Figure 8.3 Push/relabel operation for bipartite networks.

Lemma 8.5. *The bipartite preflow-push algorithm performs $O(n_1^2 m)$ nonsaturating pushes and runs in $O(n_1^2 m)$ time.*

Proof. The proof is same as that of Lemma 7.15. We consider the potential function $\Phi = \sum_{i \in I} d(i)$ whose index set I is the set of active nodes. Since we allow only the nodes in N_1 to be active, and $d(i) \leq 4n_1$ for all $i \in N_1$, the initial value of

Φ is at most $4n_1^2$. Let us observe the effect of executing the procedure bipartite push/relabel(i) on the potential function Φ. The procedure produces one of the following four outcomes: (1) it increases the distance label of node i; (2) it increases the distance label of a node $j \in N_2$; (3) it pushes flow over the arcs (i, j) and (j, k), saturating one of these two arcs; or (4) it performs a nonsaturating push. In case 1, the potential function Φ increases, but the total increase over all such iterations is only $O(n_1^2)$. In case 2, Φ remains unchanged. In case 3, Φ can increase by as much as $4n_1 + 1$ units since a new node might become active; Lemma 8.4 shows that the total increase over all iterations is $O(n_1^2 m)$. Finally, a nonsaturating push decreases the potential function by at least 2 units since it makes node i inactive, can make node k newly active and $d(k) = d(i) - 2$. This fact, in view of the preceding arguments, implies that the algorithm performs $O(n_1^2 m)$ nonsaturating pushes. Since all the other operations, such as the relabel operations and finding admissible arcs, require only $O(n_1 m)$ time, we have established the theorem. ◆

We complete this section by giving two applications of the maximum flow problem on bipartite networks with $n_1 \ll n_2$.

Application 8.1 Baseball Elimination Problem

At a particular point in the baseball season, each of $n + 1$ teams in the American League, which we number as $0, 1, \ldots, n$, has played several games. Suppose that team i has won w_i of the games that it has already played and that g_{ij} is the number of games that teams i and j have yet to play with each other. No game ends in a tie. An avid and optimistic fan of one of the teams, the Boston Red Sox, wishes to know if his team still has a chance to win the league title. We say that we can *eliminate* a specific team 0, the Red Sox, if for every possible outcome of the unplayed games, at least one team will have more wins than the Red Sox. Let $w_{\max}$ denote w_0 plus the total number of games team 0 has yet to play, which, in the best of all possible worlds, is the number of victories the Red Sox can achieve. Then we cannot eliminate team 0 if in some outcome of the remaining games to be played throughout the league, $w_{\max}$ is at least as large as the possible victories of every other team. We want to determine whether we can or cannot eliminate team 0.

We can transform this baseball elimination problem into a feasible flow problem on a bipartite network with two sets with n_1 and $n_2 = \Omega(n_1^2)$. As discussed in Section 6.2, we can represent the feasible flow problem as a maximum flow problem, as shown in Figure 8.4. The maximum flow network associated with this problem contains n *team nodes* 1 through n, $n(n - 1)/2$ *game nodes* of the type i–j for each $1 \le i \le j \le n$, and *source node* s. Each game node i–j has two incoming arcs $(i, i - j)$ and $(j, i - j)$, and the flows on these arcs represent the number of victories for team i and team j, respectively, among the additional g_{ij} games that these two teams have yet to play against each other (which is the required flow into the game node i–j). The flow x_{si} on the source arc (s, i) represents the total number of additional games that team i wins. We cannot eliminate team 0 if this network contains a feasible flow x satisfying the conditions

$$w_{\max} \ge w_i + x_{si} \qquad \text{for all } i = 1, \ldots, n,$$

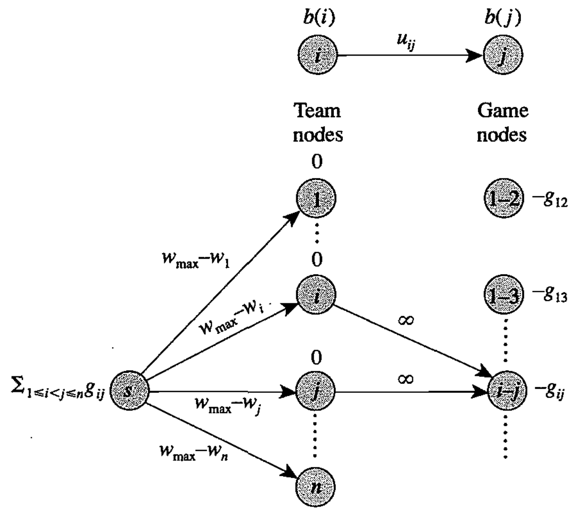


Figure 8.4 Network formulation of the baseball elimination problem.

which we can rewrite as

$$x_{si} \leq w_{\max} - w_i \qquad \text{for all } i = 1, \ldots, n.$$

This observation explains the capacities of arcs shown in the figure. We have thus shown that if the feasible flow problem shown in Figure 8.4 admits a feasible flow, we cannot eliminate team 0; otherwise, we can eliminate this team and our avid fan can turn his attention to other matters.

Application 8.2 Network Reliability Testing

In many application contexts, we need to test or monitor the arcs of a network (e.g., the tracks in a rail network) to ensure that the arcs are in good working condition. As a practical illustration, suppose that we wish to test each arc $(i, j) \in A$ in an undirected communication network $G = (N, A)$ α_{ij} times; due to resource limitations, however, each day we can test at most β_j arcs incident to any communication node $j \in N$. The problem is to find a schedule that completes the testing of all the arcs in the fewest number of days.

We solve this problem on a bipartite network $G' = (\{s\} \cup \{t\} \cup N_1 \cup N_2, A')$ defined as follows: The network contains a node $i \in N_1$ for every node $i \in N$ in the communication network and a node i–$j \in N_2$ for every arc $(i, j) \in A$ in the communication network. Each i–j node has two incoming arcs from the nodes in N_1, one from node i and the other from node j; all these arcs have infinite capacity. The source node s is connected to every node $i \in N_1$ with an arc of capacity $\lambda\beta_j$, and every node i–$j \in N_2$ is connected to the sink node t with an arc of capacity α_{ij}. The reliability testing problem is to determine the smallest integral value of the days λ so that the maximum flow in the network saturates all the sink arcs. We can solve this problem by performing binary search on λ and solving a maximum flow problem at each search point. In these maximum flow problem, $|N_1| = n$ and $|N_2| = m$, and m can be as large as $n(n - 1)/2$.

8.4 FLOWS IN PLANAR UNDIRECTED NETWORKS

A network is said to be *planar* if we can draw it in a two-dimensional (Euclidean) plane so that no two arcs cross (or intersect each other); that is, we allow the arcs to touch one another only at the nodes. Planar networks are an important special class of networks that arise in several application contexts. Because of the special structure of planar networks, network flow algorithms often run faster on these networks than they do on more general networks. Indeed, several network optimization problems are NP-complete on general networks (e.g., the maximum cut problem) but can be solved in polynomial time on planar networks. In this section we study some properties of planar networks and describe an algorithm that solves a maximum flow problem in planar networks in $O(n \log n)$ time. In this section we restrict our attention to undirected networks. We remind the reader that the undirected networks we consider contain at most one arc between any pair i and j of nodes. The capacity u_{ij} of arc (i, j) denotes the maximum amount that can flow from node i to node j or from node j to node i.

Figure 8.5 gives some examples of planar networks. The network shown in Figure 8.5(a) does not appear to be planar because arcs (1, 3) and (2, 4) cross one

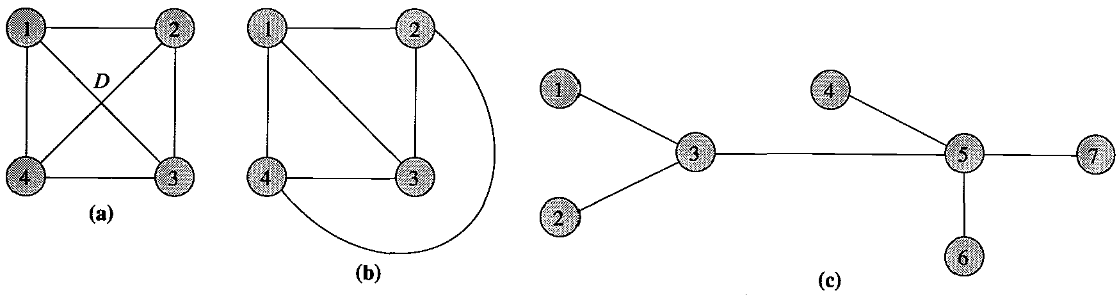


Figure 8.5 Instances of planar networks.

another at the point D, which is not a node. But, in fact, the network is planar because, as shown in Figure 8.5(b), we can redraw it, maintaining the network structure (i.e., node, arc structure), so that the arcs do not cross. For some networks, however, no matter how we draw them, some arcs will always cross. We refer to such networks as *nonplanar*. Figure 8.6 gives two instances of nonplanar networks. In both instances we could draw all but one arc without any arcs intersecting; if we add the last arc, though, at least one intersection is essential. Needless to say, determining whether a network is planar or not a straightforward task. However,

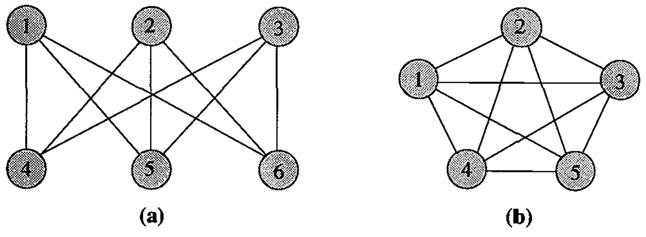


Figure 8.6 Instances of two nonplanar graphs.

researchers have developed very efficient algorithms (in fact, linear time algorithms) for testing the planarity of a network. Several theoretical characterizations of planar networks are also available.

Let $G = (N, A)$ be a planar network. A *face* z of G is a region of the (two-dimensional) plane bounded by arcs that satisfies the condition that any two points in the region can be connected by a continuous curve that meets no nodes and arcs. It is possible to draw a planar graph in several ways and each such representation might have a different set of faces. The *boundary* of a face z is the set of all arcs that enclose it. It is convenient to represent the boundary of a face by a cycle. Observe that each arc in the network belongs to the boundary of at most two faces. Faces z and z' are said to be *adjacent* if their boundaries contain a common arc. If two faces touch each other only at a node, we do not consider them to be adjacent. The network shown in Figure 8.5(b) illustrates these definitions. This network has four faces. The boundaries 1–2–3–1, 1–3–4–1, and 2–4–3–2 define the first three faces of the network. The fourth face is unbounded and consists of the remaining region; its boundary is 1–2–4–1. In Figure 8.5(b) each face is adjacent to every other face. The network shown in Figure 8.5(c) is a very special type of planar network. It has one unbounded face and its boundary includes all the arcs.

Next we discuss two well-known properties of planar networks.

Property 8.6 (Euler's Formula). *If a connected planar network has n nodes, m arcs, and f faces, then* $f = m - n + 2$.

Proof. We prove this property by performing induction on the value of f. For $f = 1$, $m = n - 1$, because a connected graph with just one face (which is the unbounded face) must be a spanning tree. Now assume, inductively, that Euler's formula is valid for every graph with k or fewer faces; we prove that the formula is valid for every graph with $k + 1$ faces. Consider a graph G with $k + 1$ faces and n nodes. We select any arc (i, j) that belongs to two faces, say z_1 and z_2 (show that the network always contains such an arc!). If we delete this arc from G, the two faces z_1 and z_2 merge into a single face. The resulting graph G' has m arcs, k faces, and n nodes, and by the induction hypothesis, $k = m - n + 2$. Therefore, if we reintroduce the arc (i, j) into G', we see that $k + 1 = (m + 1) - n + 2$, so Euler's formula remains valid. We have thus completed the inductive step and established Euler's formula in general. ◆

Property 8.7. *In a planar network,* $m < 3n$.

Proof. We prove this property by contradiction. Suppose that $m \geq 3n$. Alternatively,

$$n \geq m/3. \tag{8.1}$$

We next obtain a relationship between f and m. Since the network contains no parallel arcs, the boundary of each face contains at least three arcs. Therefore, if we traverse the boundaries of all the faces one by one, we traverse at least $3f$ arcs. Now notice that we would have traversed each arc in the network at most twice because it belongs to the boundaries of at most two faces. These observations show that $3f \leq 2m$. Alternatively,

$$f \leq 2m/3. \tag{8.2}$$

Using (8.1) and (8.2) in the formula $f = m - n + 2$, we obtain

$$2 = n - m + f \leq m/3 - m + 2m/3 = 0, \tag{8.3}$$

which is a contradiction. ◆

Property 8.7 shows that every planar graph is very sparse [i.e., $m = O(n)$]. This result, by itself, improves the running times for most network flow algorithms. For instance, as shown in Section 8.5, the shortest augmenting path algorithm for the maximum flow problem, implemented using the dynamic tree data structure, runs in $O(nm \log n)$ time. For planar networks, this time bound becomes $O(n^2 \log n)$. We can, in fact, develop even better algorithms by using the special properties of planar networks. To illustrate this point, we prove some results that apply to planar networks, but not to nonplanar networks. We show that we can obtain a minimum cut and a maximum flow for any planar network in $O(n \log n)$ time by solving a shortest path problem.

Finding Minimum Cuts Using Shortest Paths

Planar networks have many special properties. In particular, every connected planar network $G = (N, A)$ has an associated "twin" planar network $G^* = (N^*, A^*)$, which we refer to as the *dual* of G. We construct the dual G^* for a given graph G as follows. We first place a node f^* inside each face f of G. Each arc in G has a corresponding arc in G^*. Every arc (i, j) in G belongs to the boundaries of either (1) two faces, say f_1 and f_2; or (2) one face, say f_1. In case 1, G^* contains the arc (f_1^*, f_2^*); in case 2, G^* contains the loop (f_1^*, f_1^*). Figure 8.7, which illustrates this construction, depicts the dual network by dashed lines.

For notational convenience, we refer to the original network G as the *primal network*. The number of nodes in the dual network equals the number of faces in the primal network, and conversely, the number of faces in the dual network equals the number of nodes in the primal network. Both the primal and dual networks have

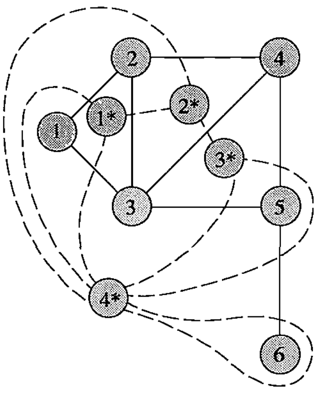


Figure 8.7 Constructing the dual of a planar network.

the same number of arcs. Furthermore, the dual of the dual network is the primal network. It is easy to show that a cycle in the dual network defines a cut in the primal network, and vice versa. For example, the cycle 4*–1*–2*–4* in the dual network shown in Figure 8.7 [with (4*, 1*) denoting the arc from 4* to 1* that also passes through arc (1, 2)], defines the cut {(2, 1), (2, 3), (2, 4)} in the primal network.

Our subsequent discussion in this section applies to a special class of planar networks known as *s–t planar* networks. A planar network with a source node *s* and a sink node *t* is called *s–t planar* if nodes *s* and *t* both lie on the boundary of the unbounded face. For example, the network shown in Figure 8.8(a) is *s–t* planar if $s = 1$ and $t = 8$; however, it is not *s–t* planar if (1) $s = 1$ and $t = 6$, or (2) $s = 3$ and $t = 8$.

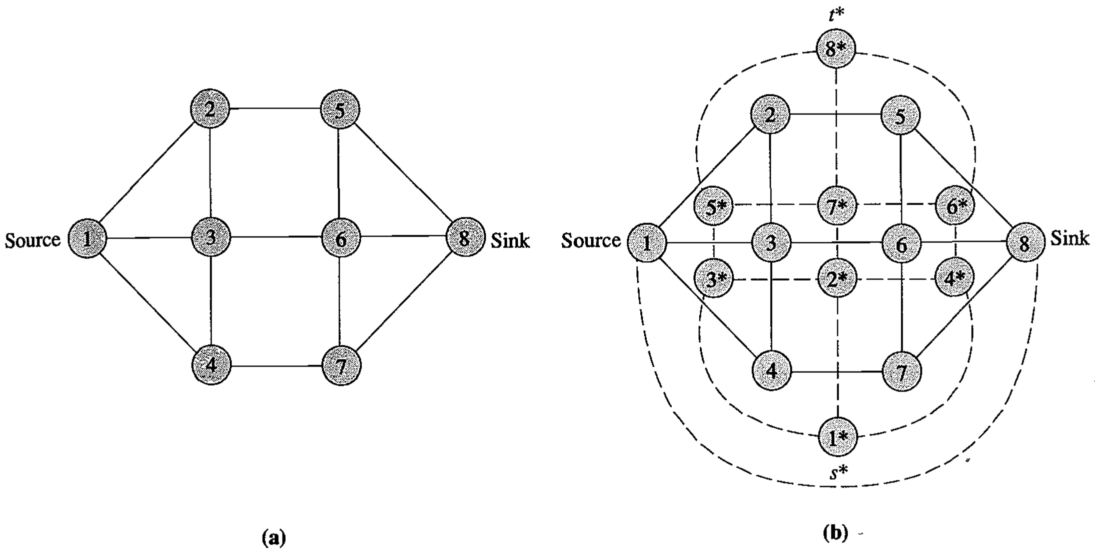


Figure 8.8 Establishing a relationship between cuts and paths: (a) *s–t* planar network; (b) corresponding dual network.

We now show how to transform a minimum cut problem on an *s–t* planar network into a shortest path problem. In the given *s–t* planar network, we first draw a new arc joining the nodes *s* and *t* so that the arc stays within the unbounded face of the network [see Figure 8.8(b)]; this construction creates a new face of the network, which we call the *additional face*, but maintains the network's planarity. We then construct the dual of this network; we designate the node corresponding to the additional face as the dual source *s** and the node corresponding to the unbounded face as the dual sink *t**. We set the cost of an arc in the dual network equal to the capacity of the corresponding arc in the primal network. The dual network contains the arc (*s**, *t**) which we delete from the network. Figure 8.8(b) shows this construction: the dashed lines are the arcs in the dual network. It is easy to establish a one-to-one correspondence between *s–t* cuts in the primal network and paths from node *s** to node *t** in the dual network; moreover, the capacity of the cut equals the cost of the corresponding path. Consequently, we can obtain a minimum s–t cut in the primal network by determining a shortest path from node s^* to node t^* in the dual network.

In the preceding discussion we showed that by solving a shortest path problem in the dual network, we can identify a minimum cut in a primal s–t planar network. Since we can solve the shortest path problem in the dual network in $O(m \log n) = O(n \log n)$ using the binary heap implementation of Dijkstra's algorithm (see Section 4.7), this development provides us with an $O(n \log n)$ algorithm for identifying a minimum cut in a planar network. Notice that this bound is substantially better than the one we would obtain for a general network. We now give a generalization of this result, obtaining a rather surprising result that the shortest path distances in the dual network provide a maximum flow in the primal network.

Let $d(j^*)$ denote the shortest path distance from node s^* to node j^* in the dual network. Recall from Section 5.2 that the shortest path distances satisfy the following conditions:

$$d(j^*) \le d(i^*) + c_{i^*j^*} \qquad \text{for each } (i^*, j^*) \in A^*. \tag{8.4}$$

Each arc (i, j) in the primal network corresponds to an arc (i^*, j^*) in the dual network. Let us define a function x_{ij} for each $(i, j) \in A$ in the following manner:

$$x_{ij} = d(j^*) - d(i^*). \tag{8.5}$$

Note that $x_{ij} = -x_{ji}$. Now notice that the network G is undirected so that the arc set A contains both the arc (i, j) and the arc (j, i). Hence we can regard a negative flow on arc (j, i) as a positive flow on arc (i, j). Consequently, the flow vector x will always nonnegative.

The expressions (8.5) and (8.4) imply that

$$x_{ij} = d(j^*) - d(i^*) \le c_{i^*j^*}. \tag{8.6}$$

Therefore, the flow x satisfies the arc capacity constraints. We next show that x also satisfies the mass balance constraints. Each node k in G, except node s and node t, defines a cut $Q = [\{k\}, N - \{k\}]$ consisting of all of the arcs incident to that node. The arcs in G^* corresponding to arcs in Q define a cycle, say W^*. For example, in Figure 8.7, the cycle corresponding to the cut for $k = 3$ is 1*–2*–3*–4*–1*. Clearly,

$$\sum_{(i^*, j^*) \in W^*} (d(j^*) - d(i^*)) = 0, \tag{8.7}$$

because the terms cancel each other. Using (8.5) in (8.7) shows that

$$\sum_{(i, j) \in Q} x_{ij} = 0,$$

which implies that inflow equals outflow at node k. Finally, we show that the flow x is a maximum flow. Let P^* be a shortest path from node s^* to node t^* in G^*. The definition of P^* implies that

$$d(j^*) - d(i^*) = c_{i^*j^*} \qquad \text{for each } (i^*, j^*) \in P^*. \tag{8.8}$$

The arcs corresponding to P^* define an s–t cut Q in the primal network. Using (8.5) in expression (8.8) and using the fact that $c_{i^*j^*} = u_{ij}$, we get

$$x_{ij} = u_{ij} \qquad \text{for each } (i, j) \in Q. \tag{8.9}$$

Consequently, the flow saturates all the arcs in an s–t cut and must be a maximum flow. The following theorem summarizes our discussion.

Theorem 8.8. *It is possible to determine a maximum flow in an s–t planar network in $O(n \log n)$ time.* ◆

8.5 DYNAMIC TREE IMPLEMENTATIONS

A dynamic tree is an important data structure that researchers have used extensively to improve the worst-case complexity of several network algorithms. In this section we describe the use of this data structure for the shortest augmenting path algorithm. We do not describe how to actually implement the dynamic tree data structure; rather, we show how to use this data structure as a "black box" to improve the computational complexity of certain algorithms. Our objective is to familiarize readers with this important data structure and enable them to use it as a black box module in the design of network algorithms.

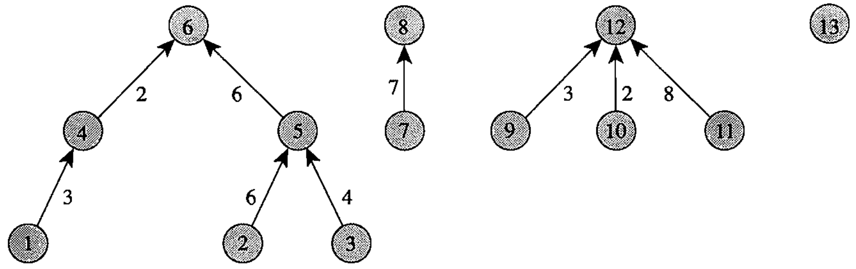
The following observation serves as a motivation for the dynamic tree structure. The shortest augmenting path algorithm repeatedly identifies a path consisting solely of admissible arcs and augments flows on these paths. Each augmentation saturates some arcs on this path, and by deleting all the saturated arcs from this path we obtain a set of *path fragments*: sets of partial paths of admissible arcs. The path fragments contain valuable information. If we reach a node in any of these path fragments using any augmenting path, we know that we can immediately extend the augmenting path along the path fragment. The standard implementation of the shortest augmenting path algorithm discards this information and possibly regenerates it again at future steps. The dynamic tree data structure cleverly stores these path fragments and uses them later to identify augmenting paths quickly.

The dynamic tree data structure maintains a collection of node-disjoint rooted trees, each arc with an associated value, called *val*. See Figure 8.9(a) for an example of the node-disjoint rooted trees. Each rooted tree is a directed in-tree with a unique root. We refer to the nodes of the tree by using the terminology of a predecessor–successor (or parent–child) relationship. For example, node 5 is the predecessor (parent) of nodes 2 and 3, and nodes 9, 10, and 11 are successors (children) of node 12. Similarly, we define the ancestors and descendants of a node (see Section 2.2 for these definitions). For example, in Figure 8.9(a) node 6 has nodes 1, 2, 3, 4, 5, 6 as its descendants, and nodes 2, 5, and 6 are the ancestors of node 2. Notice that, according to our definitions, each node is its own ancestor and descendant.

This data structure supports the following six operations:

find-root(i). Find and return the root of the tree containing node *i*.

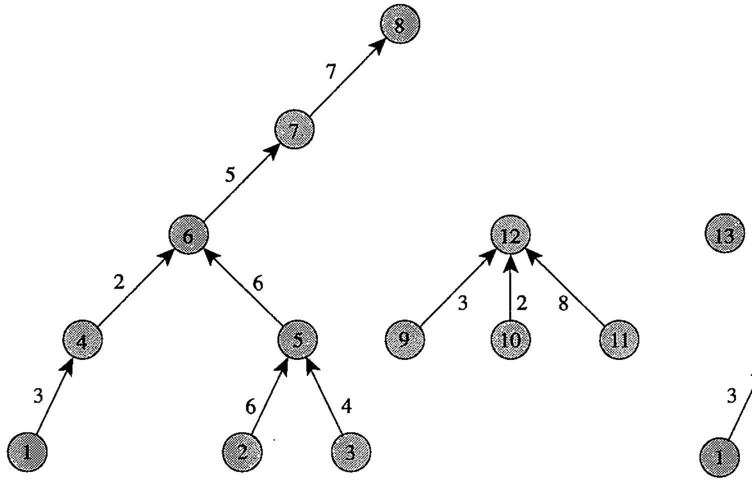
find-value(i). Find and return the value of the tree arc leaving node *i*. If *i* is a root node, return the value ∞.



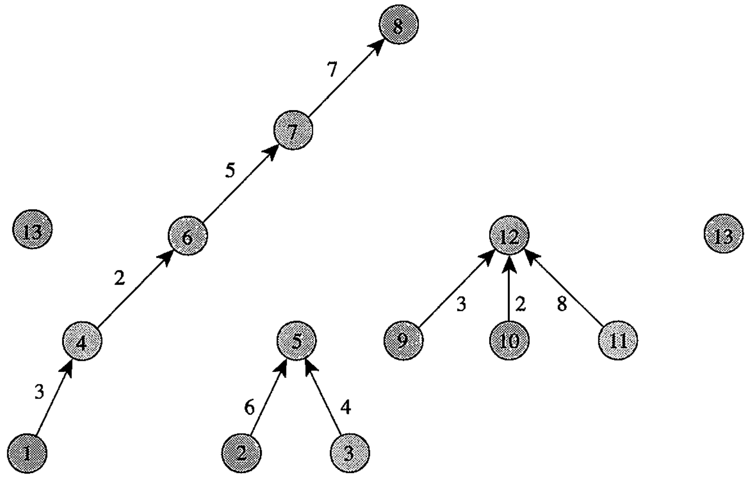
(a)

i	1	2	3	4	5	6	7	8	9	10	11	12	13
find-root(i)	6	6	6	6	6	6	8	8	12	12	12	12	13
find-value(i)	3	6	4	2	6	∞	7	∞	3	2	8	∞	∞
find-min(i)	4	5	3	4	5	6	7	8	9	10	11	12	13

(b)



(c)



(d)

Figure 8.9 Illustrating various operations on dynamic trees: (a) collection of rooted trees; (b) results of the find-root, find-value, and find-min operations; (c) rooted trees after performing link (6, 7, 5); (d) rooted trees after performing cut (5).

find-min(i). Find and return the ancestor w of i with the minimum value of find-value(w). In case of a tie, chose the node w closest to the tree root.

Figure 8.9(b) shows the results of the operations find-root(i), find-value(i), and find-min(i) performed for different nodes i.

change-value(i, val). Add a real number *val* to the value of every arc along the path from node i to find-root(i). For example, if we execute change-value(1, 3) for the dynamic tree shown in Figure 8.9(a), we add 3 to the values of arcs (1, 4) and (4, 6) and these values become 6 and 5, respectively.

link(i, j, val). This operation assumes that i is a tree root and that i and j belong to different trees. The operation combines the trees containing nodes i and j by making node j the parent of node i and giving arc (i, j) the value *val*. As an illustration, if we perform the operation link (6, 7, 5) on our example, we obtain the trees shown in Figure 8.9(c).

cut(i). Break the tree containing node i into two trees by deleting the arc joining node i to its parent and returning the value of the deleted arc. We perform this operation when i is not a tree root. For example, if we execute cut(5) on trees in Figure 8.9(c), we delete arc (5, 6) and return its value 6. Figure 8.9(d) gives the new collection of trees.

The following important result, which we state without proof, lies at the heart of the efficiency of the dynamic tree data structure.

Lemma 8.9. *If z is the maximum tree size (i.e., maximum number of nodes in any tree), a sequence of l tree operations, starting with an initial collection of singleton trees, requires a total of $O(l \log(z + l))$ time.* ◆

The dynamic tree implementation stores the values of tree arcs only implicitly. If we were to store these values explicitly, the operation change-value on a tree of size z might require $O(z)$ time (if this tree happens to be a path), which is computationally excessive for most applications. Storing the values implicitly allows us to update the values in only $O(\log z)$ time. How the values are actually stored and manipulated is beyond the scope of this book.

How might we use the dynamic tree data structure to improve the computational performance of network flow algorithms. Let us use the shortest augmenting path algorithm as an illustration. The following basic idea underlies the algorithmic speed-up. In the dynamic tree implementation, each arc in the rooted tree is an admissible arc [recall that an arc (i, j) is admissible if $r_{ij} > 0$ and $d(i) = d(j) + 1$]. The value of an arc is its residual capacity. For example, consider the residual network given in Figure 8.10(a), which shows the distance labels next to the nodes and residual capacities next to the arcs. Observe that in this network, every arc, except the arc (12, 13), is admissible; moreover, the residual capacity of every arc is 2, except for the arc (12, 14) whose residual capacity is 1. Figure 8.10(b) shows one collection of rooted trees for this example. Notice that although every tree arc is admissible, every admissible arc need not be in some tree. Consequently, for a given set of admissible arcs, many collections of rooted trees are possible.

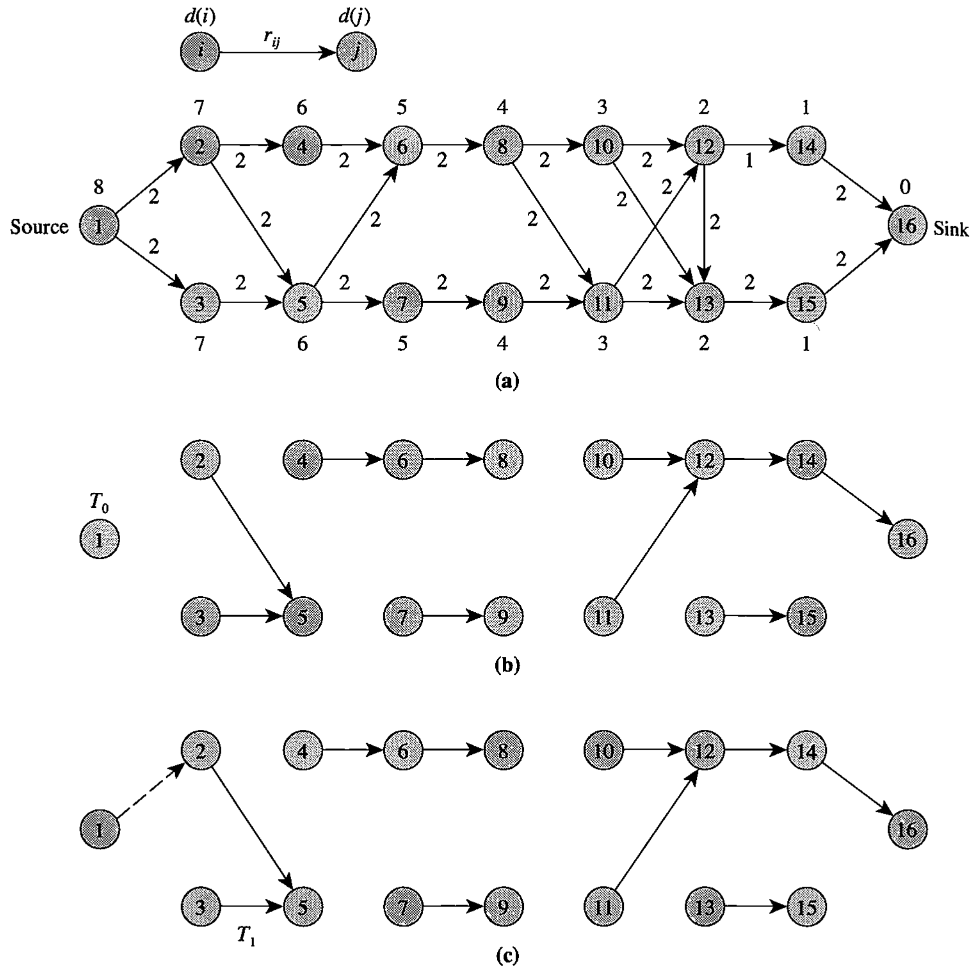
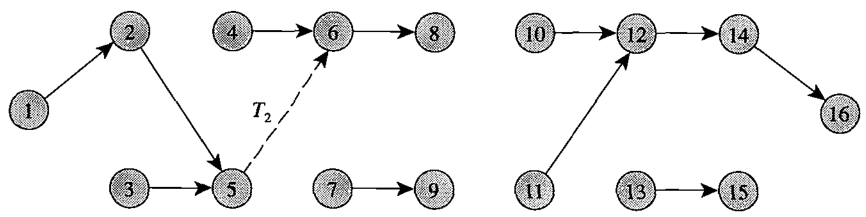
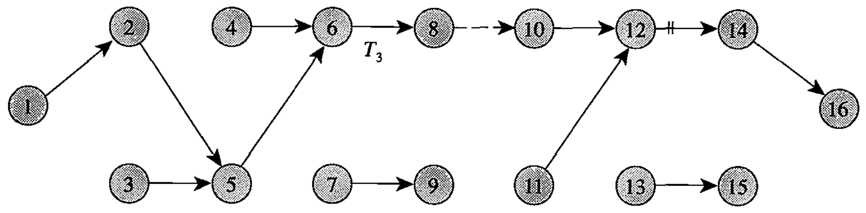


Figure 8.10 Finding an augmenting path in the dynamic tree implementations.

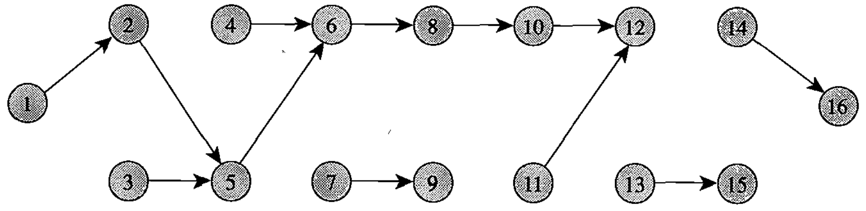
Before describing this implementation formally, we first show how the algorithm works on our example. It maintains a rooted tree containing the source node and progressively expands this tree until it contains the sink node, at which point the algorithm performs an augmentation. To grow the tree containing the source, the algorithm repeatedly performs link operations. In our example, the algorithm starts with the singleton tree T_0 containing only the source node 1 [see Figure 8.10(b)]. It identifies an admissible arc emanating from node 1. Suppose that we select arc (1, 2). The algorithm performs the operation link(1, 2, 2), which joins two rooted trees, giving us a larger tree T_1 containing node 1 [see Figure 8.10(c)]. The algorithm then identifies the root of T_1, by performing the operation find-root(1), which identifies node 5. The algorithm tries to find an admissible arc emanating from node 5. Suppose that the algorithm selects the arc (5, 6). The algorithm performs the operation link(5, 6, 2) and obtains a larger tree T_2 containing node 1 [see Figure 8.10(d)]. In the next iteration, the algorithm identifies node 8 as the root of T_2. Suppose that



(d)



(e)



(f)

Figure 8.10 (*Continued*)

the algorithm selects arc (8, 10) as the admissible arc emanating from node 8. The algorithm performs the operation link(8, 10, 2) and obtains a rooted tree T_3 that contains both the source and sink nodes [see Figure 8.10(e)].

Observe that the unique path from the source to the sink in T_3 is an admissible path since by construction every arc in a rooted tree is admissible. The residual capacity of this path is the minimum value of the arcs in this path. How can we determine this value? Recall that the operation find-min(1) would determine an ancestor of node 1 with the minimum value of find-value, which is node 12 in our example. Performing find-value(12) will give us the residual capacity of this path, which is 1 in this case. We have thus discovered the possibility of augmenting 1 unit of flow along the admissible path and that arc (12, 14) is the blocking arc. We perform the augmentation by executing change-value(1, -1). This augmentation reduces the residual capacity of arc (12, 14) to zero. The arc (12, 14) now becomes inadmissible and we must drop it from the collection of rooted trees. We do so by performing cut(12). This operation gives us the collection of rooted trees shown in Figure 8.10(f).

To better understand other situations that might occur, let us execute the algorithm for one more iteration. We apply the dynamic tree algorithm starting with the collection of rooted trees given in Figure 8.10(f). Node 12 is the root of the tree containing node 1. But node 12 has no outgoing admissible arc; so we relabel node 12. This relabeling increases the distance label of node 12 to 3. Consequently, arcs (10, 12) and (11, 12) become inadmissible and we must drop them from the collection of rooted trees. We do so by performing cut(10) and cut(11), giving us the rooted trees shown in Figure 8.11(a). The algorithm again executes find-root(1) and finds node 10 as the root of the tree containing node 1. In the next two operations, the algorithm adds arcs (10, 13) and (15, 16); Figure 8.11(b) and 8.11(c) shows the corresponding trees.

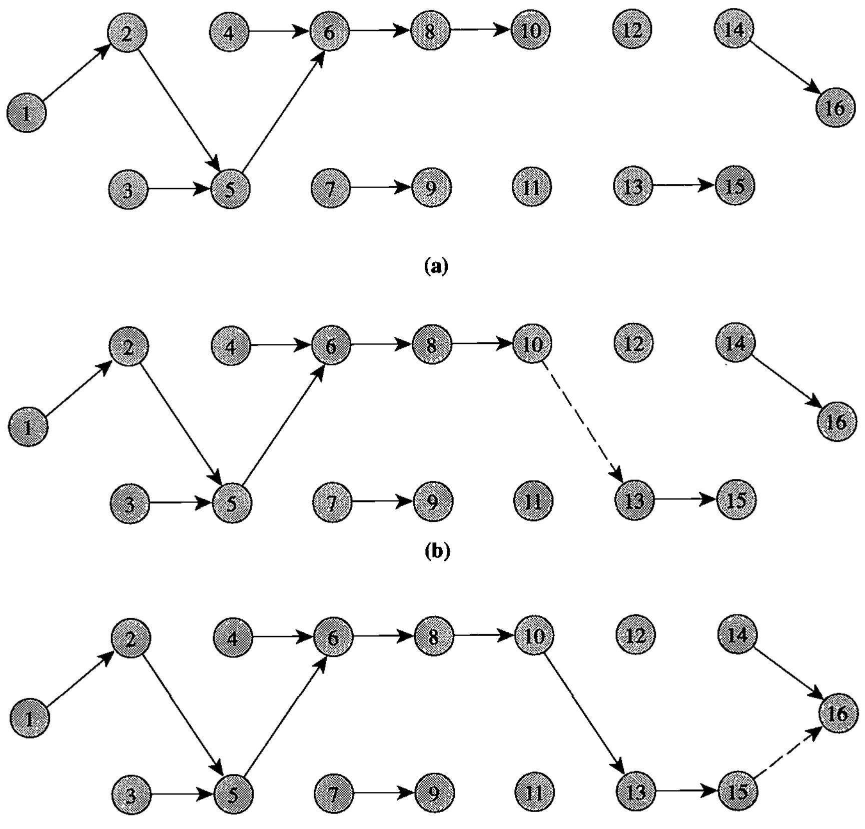


Figure 8.11 Another augmentation using dynamic trees.

Figures 8.12 and 8.13 give a formal statement of the algorithm. After offering explanatory comments, we consider a worst-case analysis of the algorithm. The algorithm is same as the one we presented in Section 7.4 except that it performs the procedures advance, retreat, and augment differently using trees. The first two procedures, tree-advance and tree-retreat, are straightforward, but the tree-augment procedure requires some explanation. If node p is an ancestor of node s with the minimum value of find-value(p) and, among such nodes in the path, it is closest to the sink, then find-value(p) gives the residual capacity of the augmenting path. The operation change(s, $-\delta$) implicitly updates the residual capacities of all the arcs in

```
algorithm tree-augmenting-path;
begin
    x : = 0;
    perform a reverse breadth-first search of the residual network
        from node t to obtain the distance labels d(i);
    let T be the collection of all singleton nodes;
    i : = s;
    while d(s) < n do
    begin
        if i has an admissible arc then tree-advance(i)
        else tree-retreat(i);
        if i = t then tree-augment;
    end;
end;
```

Figure 8.12 Dynamic tree implementation of the shortest augmenting path algorithm.

the augmenting path. This augmentation might cause the capacity of more than one arc in the path to become zero. The **while** loop identifies all such arcs, one by one, and deletes them from the collection of rooted trees.

We now consider the worst-case complexity of the algorithm. Why is the dynamic tree implementation more efficient than the original implementation of the shortest augmenting path algorithm? The bottleneck operations in the original shortest augmenting path algorithm are the advance and augment operations, which require $O(n^2m)$ time. Each advance operation in the original algorithm adds one arc;

```
procedure tree-advance(i);
begin
    let (i, j) be an admissible arc in A(i);
    link(i, j, $r_{ij}$);
    i : = find-root(j);
end;
```

(a)

```
procedure tree-retreat(i);
begin
    d(i) : = min{d(j) + 1 : (i, j) ∈ A(i) and $r_{ij}$ > 0};
    for each tree arc (k, i) do cut(k);
    i : = find-root(s);
end;
```

(b)

```
procedure tree-augment;
begin
    p : = find-min(s);
    δ : = find-value(p);
    change-value(s, -δ);
    while find-value(p) = 0 do cut(p) and set p : = find-min(s);
    i : = find-root(s);
end;
```

(c)

Figure 8.13 Procedures of the tree-augmenting-path algorithm.

in contrast, the tree implementation adds a collection of arcs using the link operation. Thus the dynamic tree implementation substantially reduces the number of executions of the link operation. Similarly, while augmenting flow, the tree implementation augments flow over a collection of arcs by performing the operation change-value, thus again substantially reducing the number of required updates.

We now obtain a bound on the number of times the algorithm performs various tree operations. We will show that the algorithm performs each of the tree operations $O(nm)$ times. In deriving these bounds, we make use of the results of Lemma 7.9, proved in Section 7.4.

cut(j). The algorithm performs this operation during the tree-retreat and tree-augment operations. During the tree-retreat(i) operation, the algorithm might perform this operation as many times as the number of incoming arcs at node i. Since this operation relabels node i, and we can relabel a node at most n times, these operations sum to $O(n^2)$ over all nodes. Furthermore, during the tree-augment operation, we perform the cut operation for each arc saturated during an augmentation. Since the total number of arc saturations is $O(nm)$, the number of these operations sums to $O(nm)$.

link(i, j, val). Each link operation adds an arc to the collection of rooted trees. Observe that if an arc enters a rooted tree, it remains there until a cut operation deletes it from the tree. Therefore, the number of link operations is at most $(n - 1)$ plus the number of cut operations. The term $(n - 1)$ arises because initially the collection might contain no arc, and finally, it might contain as many as $(n - 1)$ arcs. Consequently, the total number of link operations is also $O(nm)$. Since each tree-advance operation performs a link operation, the previous result also implies an $O(nm)$ bound on the total number of tree-advance operations.

change-value(i, val). The algorithm performs this operation once per augmentation. Since the number of augmentations is at most $nm/2$, we immediately obtain a bound of $O(nm)$ on the number of change-value operations.

find-value(i) and find-min(i). The algorithm performs each of these two operations once per augmentation and once for each arc saturated during the augmentation. These observations imply a bound of $O(nm)$ on the number of executions of these two operations.

find-root(i). The algorithm performs this operation once during each execution of the tree-advance, tree-augment, and tree-retreat operations. Since the algorithm executes the first two operations $O(nm)$ times and the third operation $O(n^2)$ times, it executes the find-root operation $O(nm)$ times.

Using simple arguments, we have now shown that the algorithm performs each of the six tree operations $O(nm)$ times. It performs each tree operation on a tree of maximum size n. The use of Lemma 8.9 establishes the following important result.

Theorem 8.10. *The dynamic tree implementation of the shortest augmenting path algorithm solves the maximum flow problem in $O(nm \log n)$ time.* ◆

Although this result establishes the theoretical utility of the dynamic tree data structure for improving the worst-case complexity of the shortest augmenting path algorithm, the practical value of this data structure is doubtful. The dynamic tree implementation reduces the time for performing advance and augment operations from $O(n^2m)$ to $O(nm \log n)$, but simultaneously increases the time of performing retreat operations from $O(nm)$ to $O(nm \log n)$. Empirical experience shows that the retreat operation is one of the bottleneck operations in practice. Since the dynamic tree data structure increases the running time of a bottleneck operation, the use of this data structure actually slows down the algorithm in practice. Furthermore, this data structure introduces substantial overhead (i.e., a large-constant factor of work is associated with each tree operation), thus making it of limited practical utility.

8.6 NETWORK CONNECTIVITY

The connectivity of a network is an important measure of the network's reliability or stability. The *arc connectivity* of a connected network is the minimum number of arcs whose removal from the network disconnects it into two or more components. In this section we suggest algorithms for solving the arc connectivity problem on undirected networks. The algorithms for solving connectivity problems on directed networks are different from those we will be discussing; we consider these algorithms in the exercises for this chapter. To conform with this choice of coverage, in this section by the term "network" we will invariably mean an undirected (connected) network.

The *node connectivity* of a network equals the minimum number of nodes whose deletion from the network disconnects it into two or more components. We discuss issues related to node connectivity in Exercise 8.35. We begin by defining several terms. A *disconnecting set* is a set of arcs whose deletion from the network disconnects it into two or more components. Therefore, the arc connectivity of a network equals the minimum cardinality of any disconnecting set; we refer to this set of arcs as a *minimum disconnecting set.*

The arc connectivity of a pair of nodes i and j is the minimum number of arcs whose removal from the network disconnects these two nodes. We represent the pair of nodes i and j by $[i, j]$ and the arc connectivity of this pair by $\alpha[i, j]$. We also let $\alpha(G)$ denote the arc connectivity of the network G. Consequently,

$$\alpha(G) = \min\{\alpha[i, j] : [i, j] \in N \times N\}. \tag{8.10}$$

We first bring together some elementary facts concerning the arc connectivity of a network; we ask the reader to prove these properties in Exercise 8.29.

Property 8.11.

(a) $\alpha[i, j] = \alpha[j, i]$ *for every pair of nodes* $[i, j]$.

(b) *The arc connectivity of a network cannot exceed the minimum degree of nodes in the network. Therefore,* $\alpha(G) \leq \lfloor m/n \rfloor$.

(c) *Any minimum disconnecting set partitions the network into exactly two components.*
(d) *The arc connectivity of a spanning tree equals* 1.
(e) *The arc connectivity of a cycle equals* 2.

Let δ denote the minimum degree of a node in the network and let node p be a node with degree equal to δ. Property 8.11(b) implies that $\alpha(G) \le \delta \le \lfloor m/n \rfloor$. Since a minimum disconnecting set partitions the node set into exactly two components $S^* \subset N$ and $\bar{S}^* = N - S^*$, we can represent this cut by the notation $[S^*, \bar{S}^*]$. We assume, without any loss of generality, that node $p \in S^*$.

Our development in Chapter 6 provides us with a means for determining the arc connectivity of a network. Theorem 6.7 states that the minimum number of arcs in a network whose removal disconnects a specified pair of source and sink nodes equals the maximum number of arc-disjoint paths from the source to the sink. Furthermore, the proof of this theorem shows that we can obtain the maximum number of arc-disjoint paths from the source to the sink by solving a maximum flow problem in a network G whose arcs all have capacity equal to 1. Thus, to determine the arc connectivity of a network, we need to solve a unit capacity maximum flow problem between every pair of nodes (by varying the source and sink nodes); the minimum value among such flows is $\alpha(G)$. Since solving a unit capacity maximum flow problem requires $O(\min\{n^{2/3}m, m^{3/2}\})$ time (see Section 8.2), this approach produces an algorithm running in time $O(n^2 \min\{n^{2/3}m, m^{3/2}\})$.

We can easily improve on this approach by a factor of n using the following idea. Consider a node $k \in \bar{S}^*$ and recall that node $p \in S^*$. Since the cut $[S^*, \bar{S}^*]$ disconnects nodes p and k, the minimum cardinality of a set of arcs that will disconnect these two nodes is at most $|\,[S^*, \bar{S}^*]\,|$. That is,

$$\alpha[p, k] \le |\,[S^*, \bar{S}^*]\,|. \tag{8.11}$$

Next observe that $[S^*, \bar{S}^*]$ is a minimum disconnecting set of the network. The definition (8.10) of $\alpha(G)$ implies that

$$\alpha[p, k] \ge |\,[S^*, \bar{S}^*]\,|. \tag{8.12}$$

Using (8.11) and (8.12), we see that

$$\alpha[p, k] = |\,[S^*, \bar{S}^*]\,|. \tag{8.13}$$

The preceding observations imply that if we compute $\alpha[p, j]$ for all j, the minimum among these numbers equals $\alpha(G)$. To summarize the discussion, we can write

$$\alpha(G) = \min\{\alpha[p, j] : j \in N - \{p\}\}.$$

The preceding approach permits us to determine the arc connectivity of a network by solving $(n - 1)$ unit capacity maximum flow problems, requiring $O(\min\{n^{5/3}m, nm^{3/2}\})$ time. We can improve this time bound for sparse networks by solving these maximum flow problems using the labeling algorithm described in Section 6.5 instead of the specialized unit capacity algorithms described in Section 8.2. The labeling algorithm will perform at most $\lfloor m/n \rfloor$ augmentations to solve each maximum flow problem (because the degree of node p is $\delta \le \lfloor m/n \rfloor$) and would require $O(m^2/n)$ time. This approach requires $O(m^2)$ time to solve all the maximum flow problems. Since $nm^{3/2} \geq m^2$, we can determine the arc connectivity of a network in $O(\min\{n^{5/3}m, m^2\})$ time. This algorithm is by no means the best algorithm for determining the arc connectivity of a network. We next describe an algorithm that computes arc connectivity in only $O(nm)$ time.

Just as the preceding algorithm determines the minimum cardinality of a set of arcs between pairs of nodes, the improved algorithm determines the minimum cardinality of a set of arcs that disconnects every node in a set S from some node $k \in \bar{S}$; we denote this number by $\alpha[S, k]$. We can compute $\alpha[S, k]$ using the labeling algorithm as follows: We allow the augmentation to start at any node in S but end only at node k. When the labeling algorithm terminates, the network contains no directed path from any node in S to node k. At this point the set of labeled nodes defines a cut in the network and the number of forward arcs in the cut is $\alpha[S, k]$.

Our preceding algorithm determines the arc connectivity of a network by computing $\alpha[p, j]$ for each node $j \in N - \{p\}$ and taking the minimum of these numbers. The correctness of this approach uses the fact that $\alpha(G)$ equals $\alpha[p, j]$ for some choice of the nodes p and j. Our improved algorithm determines the arc connectivity of a network by computing $\alpha[S, k]$ for at most $(n - 1)$ combinations of S and k and taking the minimum of these numbers. The algorithm selects the combinations S and k quite cleverly so that (1) for at least one combination of S and k, $\alpha[S, k] = \alpha(G)$; and (2) the labeling algorithm can compute $\alpha[S, k]$ for every combination in an average of $O(m)$ time because most augmentations involve only two arcs. Therefore this algorithm determines the arc connectivity of a network in $O(nm)$ time.

Before describing the algorithm, we first introduce some notation. For any set S of nodes, we let *neighbor*(S) denote the set of nodes in $\bar{S}$ that are adjacent to some node in S, and *nonneighbor*(S) as the set of nodes in $\bar{S}$ that are not adjacent to any node in S. Consequently, $N = S \cup \text{neighbor}(S) \cup \text{nonneighbor}(S)$. Our improved arc connectivity algorithm depends on the following crucial result.

Lemma 8.12. *Let δ be the minimum node degree of a network G and let $[S^*, \bar{S}^*]$ denote a minimum disconnecting set of the network. Suppose that $\alpha(G) \leq \delta - 1$. Then for any set $S \subseteq S^*$, nonneighbor(S) is nonempty.*

Proof. We first notice that the maximum number of arcs emanating from nodes in $\bar{S}^*$ is $|\bar{S}^*|(|\bar{S}^*| - 1) + \alpha(G)$ because any such arc either has both its endpoints in $\bar{S}^*$ or belongs to the minimum disconnecting set. Next notice that the minimum number of arcs emanating from the nodes in $\bar{S}^*$ is $\delta|\bar{S}^*|$ because δ is the minimum node degree. Therefore,

$$|\bar{S}^*|(|\bar{S}^*| - 1) + \alpha(G) \geq |\bar{S}^*|\delta.$$

Adding δ to both the sides of this inequality and simplifying the expression gives

$$(|\bar{S}^*| - 1)(|\bar{S}^*| - \delta) \geq \delta - \alpha(G) \geq 1.$$

The last inequality in this expression follows from the fact $\alpha(G) \leq \delta - 1$. Notice that the inequality $(|\bar{S}^*| - 1)(|\bar{S}^*| - \delta) \geq 1$ implies that both the terms to the left are at least one. Thus $|\bar{S}^*| \geq \delta + 1$; that is, the set $\bar{S}^*$ contains at least $\delta + 1$ nodes. Since the cut $[S^*, \bar{S}^*]$ contains fewer than δ arcs, at least one of the nodes in $\bar{S}^*$ is not adjacent to any node in S^*. Consequently, the set nonneighbor(S) must be nonempty, which establishes the lemma. ◆

The improved arc connectivity algorithm works as follows. It starts with $S = \{p\}$, selects a node $k \in$ nonneighbor(S), and computes $\alpha[S, k]$. It then adds node k to S, updates the sets neighbor(S) and nonneighbor(S), selects another node $k \in$ nonneighbor(S) and computes $\alpha[S, k]$. It repeats this operation until the set nonneighbor(S) is empty. The minimum value of $\alpha[S, k]$, obtained over all the iterations, is $\alpha(G)$. Figure 8.14 gives a formal description of this algorithm.

```
algorithm arc connectivity;
begin
    let $p$ be a minimum degree node in the network and $\delta$ be its degree;
    set $S^* := \{p\}$ and $\alpha^* := \delta$;
    set $S := \{p\}$;
    initialize neighbor($S$) and nonneighbor($S$);
    while nonneighbor($S$) is nonempty do
    begin
        select a node $k \in$ nonneighbor($S$);
        compute $\alpha[S, k]$ using the labeling algorithm for the maximum flow
            problem and let $[R, \bar{R}]$ be the corresponding disconnecting cut;
        if $\alpha^* > \alpha[S, k]$ then set $\alpha^* := \alpha[S, k]$ and $[S^*, \bar{S}^*] := [R, \bar{R}]$;
        add node $k$ to $S$ and update neighbor($S$), nonneighbor($S$);
    end;
end;
```

Figure 8.14 Arc connectivity algorithm.

To establish the correctness of the arc connectivity algorithm, let $[S^*, \bar{S}^*]$ denote the minimum disconnecting set. We consider two cases: when $\alpha(G) = \delta$ and when $\alpha(G) \leq \delta - 1$. If $\alpha(G) = \delta$, the algorithmic description in Figure 8.14 implies that the algorithm would terminate with $[p, N - \{p\}]$ as the minimum disconnecting set. Now suppose $\alpha(G) \leq \delta - 1$. During its execution, the arc connectivity algorithm determines $\alpha[S, k]$ for different combinations of S and k; we need to show that at some iteration, $\alpha(S, k)$ would equal $\alpha(G)$. We establish this result by proving that at some iteration, $S \subseteq S^*$ and $k \in \bar{S}^*$, in which case $\alpha[S, k] = \alpha(G)$ because the cut $[S^*, \bar{S}^*]$ disconnects every node in S from node k. Notice that initially S^* contains S (because both start with p as their only element), and finally it does not because, from Lemma 8.9, as long as S^* contains S, nonneighbor(S) is nonempty and the algorithm can add nodes to S. Now consider the last iteration for which $S \subseteq S^*$. At this iteration, the algorithm selects a node k that must be in $\bar{S}^*$ because $S \cup \{k\} \not\subseteq S^*$. But then $\alpha[S, k] = \alpha(G)$ because the cut $[S^*, \bar{S}^*]$ disconnects S from node k. This conclusion shows that the arc connectivity algorithm correctly solves the connectivity problem.

We next analyze the complexity of the arc connectivity algorithm. The algorithm uses the labeling algorithm described in Section 6.5 to compute $\alpha[S, k]$; suppose that the labeling algorithm examines labeled nodes in the first-in, first-out order so that it augments flow along shortest paths in the residual network. Each augmenting path starts at a node in S, terminates at the node k, and is one of two types: Its last internal node is in neighbor(S) or it is in nonneighbor(S). All augmenting paths of the first type are of length 2; within an iteration, we can find any such path in a total of $O(n)$ time (why?), so augmentations of the first type require a total of $O(n^2)$ time in all iterations. Detecting an augmenting path of the second type requires

$O(m)$ time; it is possible to show, however, that in all the applications of the labeling algorithm in various iterations, we never encounter more than n such augmenting paths. To see this, consider an augmenting path of the second type which contains node $l \in$ nonneighbor(S) as the last internal node in the path. At the end of this iteration, the algorithm will add node k to S; as a result, it adds node l to neighbor(S); the node will stay there until the algorithm terminates. So each time the algorithm performs an augmentation of the second type, it moves a node from the set nonneighbor(S) to neighbor(S). Consequently, the algorithm performs at most n augmentations of the second type and the total time for these augmentations will be $O(nm)$. The following theorem summarizes this discussion.

Theorem 8.13. *In $O(nm)$ time the arc connectivity algorithm correctly determines the arc connectivity of a network.* ◆

8.7 ALL-PAIRS MINIMUM VALUE CUT PROBLEM

In this section we study the all-pairs minimum value cut problem in undirected network, which is defined in the following manner. For a specific pair of nodes i and j, we define an $[i, j]$ *cut* as a set of arcs whose deletion from the network disconnects the network into two components S_{ij} and $\bar{S}_{ij}$ so that nodes i and j belong to different components (i.e., if $i \in S_{ij}$, then $j \in \bar{S}_{ij}$; and if $i \in \bar{S}_{ij}$, then $j \in S_{ij}$). We refer to this $[i, j]$ cut as $[S_{ij}, \bar{S}_{ij}]$ and say that this cut *separates* nodes i and j. We associate with a cut $[S_{ij}, \bar{S}_{ij}]$, a *value* that is a function of arcs in the cut. A *minimum* $[i, j]$ *cut* is a cut whose value is minimum among all $[i, j]$ cuts. We let $[S^*_{ij}, \bar{S}^*_{ij}]$ denote a minimum value $[i, j]$ cut and let $v[i, j]$ denote its value. The all-pairs minimum value cut problem requires us to determine for all pairs of nodes i and j, a minimum value $[i, j]$ cut $[S^*_{ij}, \bar{S}^*_{ij}]$ and its value $v[i, j]$.

The definition of a cut implies that if $[S_{ij}, \bar{S}_{ij}]$ is an $[i, j]$ cut, it is also a $[j, i]$ cut. Therefore, $v[i, j] = v[j, i]$ for all pairs i and j of nodes. This observation implies that we can solve the all-pairs minimum value cut problem by invoking $n(n - 1)/2$ applications of any algorithm for the single pair minimum value cut problem. We can, however, do better. In this section we show that we can solve the all-pairs minimum value cut problem by invoking only $(n - 1)$ applications of the single-pair minimum value cut problem.

We first mention some specializations of the all-pairs minimum value cut problem on undirected networks. If we define the value of a cut as its capacity (i.e., the sum of capacities of arcs in the cut), the all-pairs minimum value cut problem would identify minimum cuts (as defined in Chapter 6) between all pairs of nodes. Since the minimum cut capacity equals the maximum flow value, we also obtain the maximum flow values between all pairs of nodes. Several other functions defined on a cut $[S_{ij}, \bar{S}_{ij}]$ are plausible, including (1) the number of arcs in the cut, (2) the capacity of the cut divided by the number of arcs in the cut, and (3) the capacity of the cut divided by $|S_{ij}| \, |\bar{S}_{ij}|$.

We first state and prove an elementary lemma concerning minimum value cuts.

Lemma 8.14. *Let $i_1, i_2, \ldots, i_k$ be an (arbitrary) sequence of nodes. Then $v[i_1, i_k] \geq \min\{v[i_1, i_2], v[i_2, i_3], \ldots, v[i_{k-1}, i_k]\}$.*

Proof. Let $i = i_1, j = i_k$, and $[S_{ij}^*, \bar{S}_{ij}^*]$ be the minimum value $[i, j]$ cut. Consider the sequence of nodes $i_1, i_2, \ldots, i_k$ in order and identify the smallest index r satisfying the property that i_r and i_{r+1} are in different components of the cut $[S_{ij}^*, \bar{S}_{ij}^*]$. Such an index must exist because, by definition, $i_1 = i \in S_{ij}^*$ and $i_k = j \notin S_{ij}^*$. Therefore, $[S_{ij}^*, \bar{S}_{ij}^*]$ is also an $[i_r, i_{r+1}]$ cut, which implies that the value of the minimum value $[i_r, i_{r+1}]$ cut will be no more than the value of the cut $[S_{ij}^*, \bar{S}_{ij}^*]$. In other words,

$$v[i_1, i_k] \geq v[i_r, i_{r+1}] \geq \min\{v[i_1, i_2], v[i_2, i_3], \ldots, v[i_{k-1}, i_k]\}, \qquad (8.14)$$

which is the desired conclusion of the lemma. ◆

Lemma 8.14 has several interesting implications. Select any three nodes i, j, and k of the network and consider the minimum cut values $v[i, j]$, $v[j, k]$, and $v[k, i]$ between them. The inequality (8.14) implies that at least two of the values must be equal. For if these three values are distinct, then placing the smallest value on the left-hand side of (8.14) would contradict this inequality. Furthermore, it is possible to show that one of these values that is not equal to the other two must be the largest. Since for every three nodes, two of the three minimum cut values must be equal, it is conceivable that many of the $n(n - 1)/2$ cut values will be equal. Indeed, it is possible to show that the number of distinct minimum cut values is at most $(n - 1)$. This result is the subject of our next lemma. This lemma requires some background concerning the *maximum spanning tree problem* that we discuss in Chapter 13. In an undirected network G, with an associated value (or, profit) for each arc, the maximum spanning tree problem seeks a spanning tree T^*, from among all spanning trees, with the largest sum of the values of its arcs. In Theorem 13.4 we state the following optimality condition for the maximum spanning tree problem: A spanning tree T^* is a maximum spanning tree if and only if for every nontree arc (k, l), the value of the arc (k, l) is less than or equal to the value of every arc in the unique tree path from node k to node l.

Lemma 8.15. *In the $n(n - 1)/2$ minimum cut values between all pairs of nodes, at most $(n - 1)$ values are distinct.*

Proof. We construct a *complete* undirected graph $G' = (N, A')$ with n nodes. We set the value of each arc $(i, j) \in A'$ equal to $v[i, j]$ and associate the cut $[S_{ij}^*, \bar{S}_{ij}^*]$ with this arc. Let T^* be a maximum spanning tree of G'. Clearly, $|T^*| = n - 1$. We shall prove that the value of every nontree arc is equal to the value of some tree arc in T^* and this result would imply the conclusion of the lemma.

Consider a nontree arc (k, l) of value $v[k, l]$. Let P denote the unique path in T^* between nodes k and l. The fact that T^* is a maximum spanning tree implies that the value of arc (k, l) is less than or equal to the value of every arc $(i, j) \in P$. Therefore,

$$v[k, l] \leq \min[v[i, j] : (i, j) \in P]. \qquad (8.15)$$

Now consider the sequence of nodes in P that starts at node k and ends at node l. Lemma 8.14 implies that

$$v[k, l] \geq \min[v[i, j] : (i, j) \in P]. \qquad (8.16)$$

The inequalities (8.15) and (8.16) together imply that

$$v[k, l] = \min[v[i, j]:(i, j) \in P].$$

Consequently, the value of arc (k, l) equals the minimum value of an arc in P, which completes the proof of the lemma. ◆

The preceding lemma implies that we can store the $n(n - 1)/2$ minimum cut values in the form of a spanning tree T^* with a cut value associated with every arc in the tree. To determine the minimum cut values $v[k, l]$ between a pair k and l of nodes, we simply traverse the tree path from node k to node l; the cut value $v[k, l]$ equals the minimum value of any arc encountered in this path. Note that the preceding lemma only establishes the fact that there are at most $(n - 1)$ distinct minimum cut values and shows how to store them compactly in the form of a spanning tree. It does not, however, tell us whether we can determine these distinct cut values by solving $(n - 1)$ minimum cut problems, because the proof of the lemma requires the availability of minimum cut values between all node pairs which we do not have.

Now, we ask a related question. Just as we can concisely store the minimum cut values between all pairs of nodes by storing only $(n - 1)$ values, can we also store the minimum value *cuts* between all pairs of nodes concisely by storing only $(n - 1)$ cuts? Because the network has at most $(n - 1)$ distinct minimum cut values between $n(n - 1)/2$ pairs of nodes, does it have at most $(n - 1)$ distinct cuts that define the minimum cuts between all node pairs? In the following discussion we provide an affirmative answer to this question. Consider a pair k and l of nodes. Suppose that arc (i, j) is a minimum value arc in the path from node k to node l in T^*. Our preceding observations imply that $v[k, l] = v[i, j]$. Also notice that we have associated a cut $[S_{ij}^*, \bar{S}_{ij}^*]$ of value $v[i, j] = v[k, l]$ with the arc (i, j); this cut separates nodes i and j. Is $[S_{ij}^*, \bar{S}_{ij}^*]$ a minimum $[k, l]$ cut? It is if $[S_{ij}^*, \bar{S}_{ij}^*]$ separates nodes k and l, and it is not otherwise. If, indeed, $[S_{ij}^*, \bar{S}_{ij}^*]$ separates nodes k and l, and if the same result is true for every pair of nodes in the network, the cuts associated with arcs in T^* concisely store minimum value cuts between all pairs of nodes. We refer to such a tree T^* as a *separator tree*. In this section we show that every network G has a separator tree and that we can construct the separator tree by evaluating $(n - 1)$ single-pair minimum cut values. Before we describe this method, we restate the definition of the separator tree for easy future reference.

Separator tree. *An undirected spanning tree T^*, with a minimum $[i, j]$ cut $[S_{ij}^*, \bar{S}_{ij}^*]$ of value $v[i, j]$ associated with each arc (i, j), is a separator tree if it satisfies the following property for every nontree arc (k, l): If arc (i, j) is the minimum value arc from node k to node l in T^* (breaking ties in a manner to be described later), $[S_{ij}^*, \bar{S}_{ij}^*]$ separates nodes k and l.*

Our preceding discussion shows that we have reduced the all-pairs minimum value cut problem to a problem of obtaining a separator tree. Given the separator tree T^*, we determine the minimum $[k, l]$ cut as follows: We traverse the tree path from node k to node l; the cut corresponding to the minimum value in the path (breaking ties appropriately) is a minimum $[k, l]$ cut.

We call a subtree of a separator tree a *separator subtree*. Our algorithm for constructing a separator tree proceeds by constructing a separator subtree that spans an expanding set for nodes. It starts with a singleton node, adds one additional node to the separator subtree at every iteration, and terminates when the separator tree spans all the nodes. We add nodes to the separator subtree in the order 1, 2, 3, . . . , n. Let T^{p-1} denote the separator subtree for the node set $\{1, 2, \ldots, p-1\}$ and T^p denote the separator subtree for the node set $\{1, 2, \ldots, p\}$. We obtain T^p from T^{p-1} by adding an arc, say (p, k). The essential problem is to locate the node k incident to node p in T^p. Once we have located the node k, we identify a minimum value cut $[S^*_{pk}, \bar{S}^*_{pk}]$ between nodes p and k, and associate it with the arc (p, k). We set the value of the arc (p, k) equal to $v[p, k]$.

As already mentioned, our algorithm for constructing the separator tree adds arcs to the separator subtree one by one. We associate an index, called an *order* index, with every arc in the separator subtree. The first arc added to the separator subtree has order index 1, the second arc added has order index 2, and so on. We use the order index to resolve ties while finding a minimum value arc between a pair of nodes. As a rule, whether we specify so or not in the subsequent discussion, we always resolve any tie in favor of the arc with the least order index (i.e., the arc that we added first to the separator subtree).

Figure 8.15 describes the procedure we use to locate the node $k \in T^{p-1}$ on which the arc (p, k) will be incident in T^p.

Note that in every iteration of the locate procedure, $T_\alpha \subseteq N_\alpha$ and $T_\beta \subseteq N_\beta$. This fact follows from the observation that for every $k \in T_\alpha$ and $l \in T_\beta$, the arc (α, β) is the minimum value arc in the path from node k to node l in the separator subtree T and, by its definition, the cut must separate node k and node l.

We illustrate the procedure locate using a numerical example. Consider a nine-node network with nodes numbered 1, 2, 3, . . . , 9. Suppose that after five iterations, the separator subtree $T^{p-1} = T^6$ is as shown in Figure 8.16(a). The figure also shows the cut associated with each arc in the separator subtree (here we specify only $S^*_{\alpha\beta}$, because we can compute $\bar{S}^*_{\alpha\beta}$ by using $\bar{S}^*_{\alpha\beta} = N - S^*_{\alpha\beta}$). We next consider adding node 7 to the subtree. At this point, $T = T^6$ and the minimum value arc in T is (4, 5). Examining S^*_{45} reveals that node 7 is on the same side of the cut as node

```
procedure locate($T^{p-1}$, $p$, $k$);
begin
    $T := T^{p-1}$;
    while $T$ is not a singleton node do
    begin
        let $(\alpha, \beta)$ be the minimum value arc in $T$ (we break ties in favor of the arc with the
            smallest order index);
        let $[S^*_{\alpha\beta}, \bar{S}^*_{\alpha\beta}]$ be the cut associated with the arc $(\alpha, \beta)$;
        let the arc $(\alpha, \beta)$ partition the tree $T$ into the subtrees $T_\alpha$
            and $T_\beta$ so that $\alpha \in T_\alpha$ and $\beta \in T_\beta$;
        let the cut $[S^*_{\alpha\beta}, \bar{S}^*_{\alpha\beta}]$ partition the node set $N$ into the
            subsets $N_\alpha$ and $N_\beta$ so that $\alpha \in N_\alpha$ and $\beta \in N_\beta$;
        if $p \in N_\alpha$ then set $T := T_\alpha$ else set $T := T_\beta$;
    end;
    set $k$ equal to the singleton node in $T$;
end;
```

Figure 8.15 Locate procedure.

4; therefore, we update T to be the subtree containing node 4. Figure 8.16(b) shows the tree T. Now arc (2, 3) is the minimum value arc in T. Examining S^*_{23} reveals that node 7 is on the same side of the cut as node 2; so we update T so that it is the subtree containing node 2. At this point, the tree T is a singleton, node 2. We set $k = 2$ and terminate the procedure. We next add arc (2, 7) to the separator subtree, obtain a minimum value cut between the nodes 7 and 2, and associate this cut with the arc (2, 7). Let $v[7, 2] = 5$ and $S^*_{72} = \{7\}$. Figure 8.16(c) shows the separator subtree spanning the nodes 1 through 7.

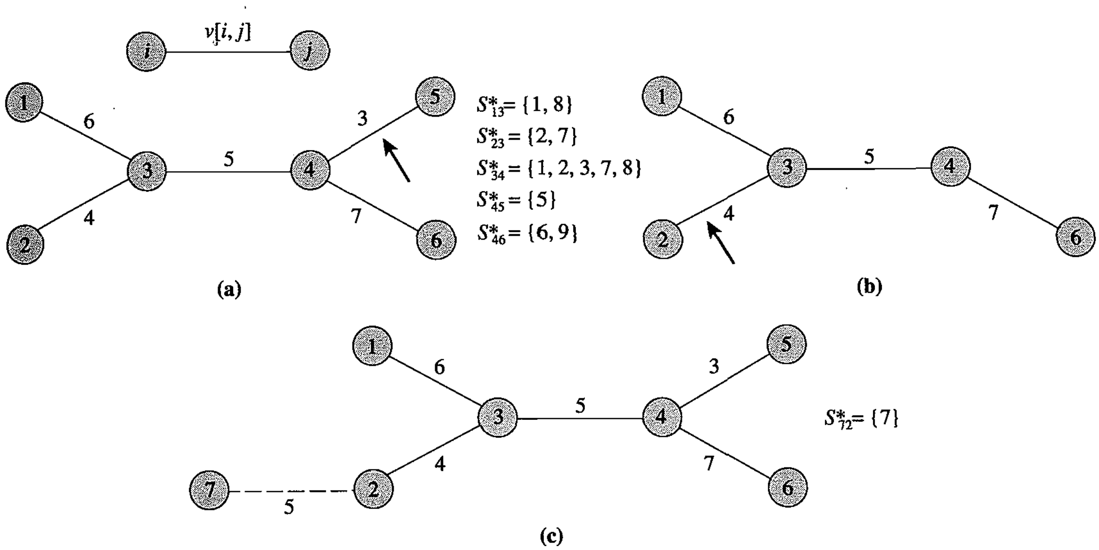


Figure 8.16 Illustrating the procedure locate.

We are now in a position to prove that the subtree T^p is a separator subtree spanning the nodes $\{1, 2, \ldots, p\}$. Since, by our inductive hypothesis, T^{p-1} is a separator subtree on the nodes $\{1, 2, \ldots, p - 1\}$, our proof amounts to establishing the following result for every node $l \in \{1, 2, \ldots, p - 1\}$: If (i, j) is the minimum value arc in T^p in the path from node p to node l (with ties broken appropriately), the cut $[S^*_{ij}, \bar{S}^*_{ij}]$ separates the nodes p and l. We prove this result in the following lemma.

Lemma 8.16. *For any node $l \in T^{p-1}$, if (i, j) is the minimum value arc in T^p in the path from node p to node l (when we break ties in favor of the arc with the least order index), $[S^*_{ij}, \bar{S}^*_{ij}]$ separates nodes p and l.*

Proof. We consider two possibilities for the arc (i, j).

Case 1: $(i, j) = (p, k)$. Let P denote the tree path in T^p from node k to node l. The situation $(i, j) = (p, k)$ can occur only when arc (p, k) is the unique minimum value arc in T^p in P, for otherwise, the tie will be broken in favor of an arc other than the arc (p, k) (why?). Thus

$$v[p, k] < v[g, h] \qquad \text{for every arc } (g, h) \in P. \tag{8.17}$$

Next consider any arc $(g, h) \in P$. We claim that both the nodes g and h must belong to the same component of the cut $[S^*_{pk}, \bar{S}^*_{pk}]$; for otherwise, $[S^*_{pk}, \bar{S}^*_{pk}]$ will also separate nodes g and h, so $v[p, k] \geq v[g, h]$, contradicting (8.17). Using this argument inductively for all arcs in P, we see that all the nodes in the path P (that starts at node k and ends at node l) must belong to the same component of the cut $[S^*_{pk}, \bar{S}^*_{pk}]$. Since the cut $[S^*_{pk}, \bar{S}^*_{pk}]$ separates nodes p and k, it also separates nodes p and l.

Case 2: $(i, j) \neq (p, k)$. We examine this case using the locate procedure. At the beginning of the locate procedure, $T = T^{p-1}$, and at every iteration the size of the tree becomes smaller, until finally, $T = \{k\}$. Consider the iteration when T contains both the nodes k and l, but in the next iteration the tree does not contain node l. Let P denote the path in T from node k to node l in this iteration. It is easy to see that the arc (α, β) selected by the locate procedure in this iteration must belong to the path P. By definition, (α, β) is the minimum value arc in T, with ties broken appropriately. Since T contains the path P, (α, β) is also a minimum value arc in P. Now notice from the statement of the lemma that arc (i, j) is defined as the minimum value arc in P, and since we break the tie in precisely the same manner, $(i, j) = (\alpha, \beta)$.

We next show that the cut $[S^*_{\alpha\beta}, \bar{S}^*_{\alpha\beta}]$ separates node p and node l. We recommend that the reader refers to Figure 8.17 while reading the remainder of the proof. Consider the same iteration of the locate procedure considered in the preceding paragraph, and let T_α, N_α, T_β, N_β be defined as in Figure 8.15. We have observed previously that $T_\alpha \subseteq N_\alpha$ and $T_\beta \subseteq N_\beta$. We assume that $p \in N_\alpha$; a similar argument applies when $p \in N_\beta$. The procedure implies that when $p \in N_\alpha$, we set $T = T_\alpha$, implying that $k \in T_\alpha \subseteq N_\alpha$. Since the cut $[S^*_{\alpha\beta}, \bar{S}^*_{\alpha\beta}]$ separates node k from node l it also separates node p from node l. The proof of the lemma is complete. ◆

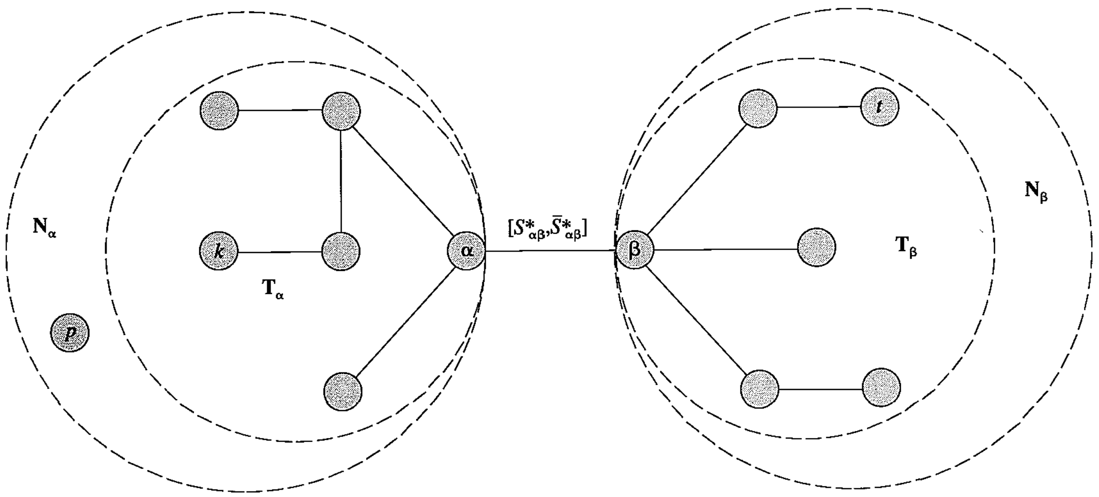


Figure 8.17 Proving Lemma 8.16.

Having proved the correctness of the all-pairs minimum cut algorithm, we next analyze its running time. The algorithm performs $(n - 1)$ iterations. In each iteration it executes the locate procedure and identifies the arc (p, k) to be added to the separator subtree. The reader can easily verify that an execution of the locate procedure requires $O(n^2)$ time. The algorithm then solves a minimum value cut problem and associates this cut and its value with the arc (p, k). The following theorem is immediate.

Theorem 8.17. *Solving the all-pairs minimum value cut problem requires $O(n^3)$ time plus the time required to solve $(n - 1)$ instances of the single-pair minimum value cut problem.* ◆

To summarize, we have shown that the all-pairs minimum cut algorithm finds the minimum capacity cut separating node i and node j (i.e., with node i on one side of the cut and node j on the other side) for every node pair $[i, j]$. The max-flow min-cut theorem shows that this algorithm also determines the maximum flow values between every pair of nodes.

Suppose, instead, that we are given a directed graph. Let $f[i, j]$ denote the value of the minimum cut from node i to node j. We cannot use the all-pairs minimum cut algorithm to determine $f[i, j]$ for all node pairs $[i, j]$ for the following simple reason: The algorithm would determine the minimum value of a cut separating node i from node j, and this value is $\min\{f[i, j], f[j, i]\}$ because the minimum cut from node i to node j separates nodes i and j and so does the minimum cut from node j to node i. (We did not face this problem for undirected networks because the minimum cut from node i to node j is also a minimum cut from node j to node i.) If we let $v[i, j] = \min\{f[i, j], f[j, i]\}$, we can use the all-pairs minimum cut algorithm to determine $v[i, j]$ for each node pair $[i, j]$. Moreover, this algorithm relies on evaluating $v[i, j]$ for only $(n - 1)$ pairs of nodes. Since we can determine $v[i, j]$ by finding a maximum flow from node i to node j and from node j to node i, we can compute $v[i, j]$ for all node pairs by solving $(2n - 2)$ maximum flow problems.

We complete this section by describing an application of the all-pairs minimum value cut problem.

Application 8.3 Maximum and Minimum Arc Flows in a Feasible Flow

Consider the feasible flow problem that we discussed in Application 6.1. Assume that the network is uncapacitated and that it admits a feasible flow. For each arc $(i, j) \in A$, let α_{ij} denote the minimum arc flow that (i, j) can have in some feasible flow, and let β_{ij} denote the maximum arc flow that (i, j) can have in some feasible flow. We will show that we can determine α_{ij} for all node pairs $[i, j]$ by solving at most n maximum flow problems, and we can determine β_{ij} for all node pairs $[i, j]$ by an application of the all-pairs minimum value cut problem.

The problem of determining the maximum and minimum values of the arc flows in a feasible flow arises in the context of determining the statistical security of data (e.g., census data). Given a two-dimensional table A of size $p \times q$, suppose that we want to disclose the row sums r_i, the column sums c_j, and a subset of the matrix elements. For security reasons (or to ensure confidentiality of the data), we would like to ensure that we have "disguised" the remaining matrix elements (or "hidden" entries). We wish to address the following question: Once we have disclosed the row and column sums and some matrix elements, how secure are the hidden entries? We address a related question: For each hidden element a_{ij}, what are the minimum and maximum values that a_{ij} can assume consistent with the data we have disclosed? If these two bounds are quite close, the element a_{ij} is not secure.

We assume that each revealed matrix element has value 0. We incur no loss of generality in making this assumption since we can replace a nonzero element a_{ij} by 0, replace r_i by $r_i - a_{ij}$, and replace c_j by $c_j - a_{ij}$. To conduct our analysis, we begin by constructing the bipartite network shown in Figure 8.18; in this network, each unrevealed matrix element a_{ij} corresponds to an arc from node i to node $\bar{j}$. It is easy to see that every feasible flow in this network gives values of the matrix elements that are consistent with the row and column sums.

How might we compute the α_{ij} values? Let x^* be any feasible flow in the network (which we can determine by solving a maximum flow problem). In Section 11.2 we show how to convert each feasible flow into a feasible spanning tree solution; in this solution at most $(n - 1)$ arcs have positive flow. As a consequence, if x^* is a spanning tree solution, at least $(m - n + 1)$ arcs have zero flow; and therefore, $\alpha_{ij} = 0$ for each of these arcs. So we need to find the α_{ij} values for only the remaining $(n - 1)$ arcs. We claim that we can accomplish this task by solving at most $(n - 1)$ maximum flow problems. Let us consider a specific arc (i, j). To determine the minimum flow on arc (i, j), we find the maximum flow from node i to node j in $G(x^*)$ when we have deleted arc (i, j). If the maximum flow from node i to node j has value k, we can reroute up to k units of flow from node i to node j and reduce the flow an arc (i, j) by the same amount. As a result, $\alpha_{ij} = \max\{0, x_{ij}^* - k\}$.

To determine the maximum possible flow on arc (i, j), we determine the maximum flow from node j to node i in $G(x^*)$. If we can send k units from node j to node i in $G(x^*)$, then $\beta_{ij} = k$. To establish this result, suppose that the k units consist of x_{ij}^* units on arc (j, i) [which is the reversal of arc (i, j)], and $k - x_{ij}^*$ units that do not use arc (i, j). Then, to determine the maximum flow on the arc (i, j), we can

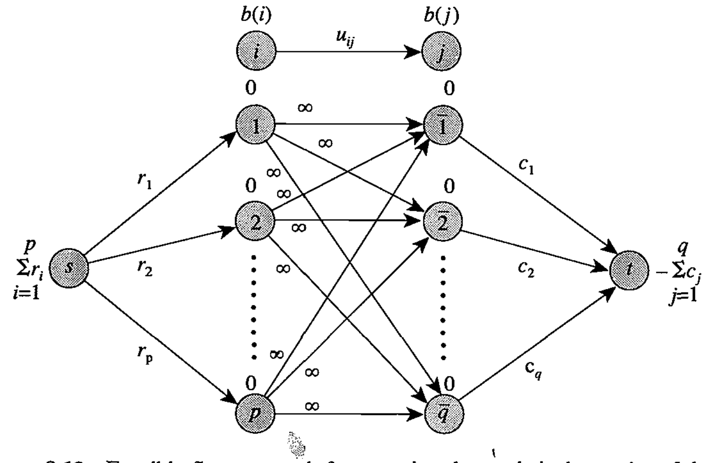


Figure 8.18 Feasible flow network for ensuring the statistical security of data.

send $k - x^*_{ij}$ units of flow from node j to node i and increase the flow in arc (i, j) by $k - x^*_{ij}$, leading to a total of k units.

Thus, to find β_{ij}, we need to compute the maximum flow from node j to node i in $G(x^*)$ for each arc $(i,j) \in A$. Equivalently, we want to find the minimum capacity cut $f[j, i]$ from node j to node i in $G(x^*)$. As stated previously, we cannot find $f[i,j]$ for all node pairs $[i,j]$ in a directed network; but we can determine $\min\{f[i,j], f[j, i]\}$. We now use the fact that we need to compute $f[j, i]$ when $(i, j) \in A$, in which case $f[i,j] = \infty$ (because the network is uncapacitated). Therefore, for each arc $(i, j) \in A$, $f[i,j] = \min\{f[i,j], f[j, i]\}$, and we can use the all-pairs minimum value cut algorithm to determine all of the β_{ij} values by solving $(2n - 2)$ maximum flow problems.

8.8 SUMMARY

As we have noted in our study of shortest path problems in our earlier chapters, we can sometimes develop more efficient algorithms by restricting the class of networks that we wish to consider (e.g., networks with nonnegative costs, or acyclic networks). In this chapter we have developed efficient special purpose algorithms for several maximum flow problems with specialized structure: (1) unit capacity networks, in which all arcs have unit capacities; (2) unit capacity simple networks, in which each node has a single incoming or outgoing arc; (3) bipartite networks; and (4) planar networks. We also considered one specialization of the maximum flow problem, the problem of finding the maximum number of arc-disjoint paths between two nodes in a network, and one generalization of the minimum cut problem, finding minimum value cuts between all pairs of nodes. For this last problem we permitted ourselves to measure the value of any cut by a function that is more general than the sum of the capacities of the arcs in the cut. Finally, we considered one other advanced topic: the use of the dynamic trees data structure to efficiently implement the shortest augmenting path algorithm for the maximum flow problem.

Figure 8.19, which summarizes the basic features of these various algorithms,

Algorithm	Ruuning time	Features
Maximum flow algorithm for unit capacity networks	$O(\min\{n^{2/3}m, m^{3/2}\})$	1. Fastest available algorithm for solving the maximum flow problem in unit capacity networks. 2. Uses two phases: first applies the shortest augmenting path algorithm until the distance label of node s satisfies the condition $d(s) \geq d^* = \min\{\lceil 2n^{2/3} \rceil, \lceil m^{1/2} \rceil\}$. At this point, uses the labeling algorithm until it establishes a maximum flow. 3. Easy to implement and is likely to be efficient in practice.
Maximum flow algorithm for unit capacity simple networks	$O(n^{1/2}m)$	1. Fastest available algorithm for solving the maximum flow problem in unit capacity simple networks. 2. Same two phase approach as the preceding algorithm, except $d^* = \lceil n^{1/2} \rceil$.

Figure 8.19 Summary of algorithms discussed in this chapter.

Algorithm	Running time	Features
Bipartite preflow-push algorithm	$O(n_1^2 m)$	1. Faster approach for solving maximum flow problems in bipartite networks satisfying the condition $n_1 < n_2$. 2. Improved implementation of the generic preflow-push algorithm discussed in Section 7.6. 3. Uses "two-arc" push rule in which we always push flow from an active node over two consecutive admissible arcs. 4. As discussed in the exercises, significant further improvements are possible if we examine active nodes in some specific order.
Planar maximum flow algorithm	$O(n \log n)$	1. Highly efficient algorithm for solving the maximum flow problem in s–t planar networks. 2. Constructs the dual network and solves a shortest path problem over it. The shortest path in the dual network yields a minimum cut in the original network and the shortest path distances yield a maximum flow. 3. Applicable only to undirected s–t planar networks.
Dynamic tree algorithm	$O(nm \log n)$	1. Uses the dynamic tree data structure to implement the shortest augmenting path algorithm for the maximum flow problem. 2. Improves the running time of the shortest augmenting path algorithm from $O(n^2m)$ to $O(nm \log n)$. 3. Similar, though not as dramatic, improvements can be obtained by using this data structure in preflow-push algorithms. 4. The dynamic tree data structure is quite sophisticated, has substantial overhead and its practical usefulness has not yet been established.
Arc connectivity algorithm	$O(nm)$	1. Fastest available algorithm for obtaining the arc connectivity of a network. 2. Uses the labeling algorithm for the maximum flow problem as a subroutine. 3. Likely to be very efficient in practice. 4. Applicable only to undirected networks.
All-pairs minimum cut algorithm	$O(nM(n, m, U) + n^3)$	1. Fastest available algorithm for solving the all-pairs minimum cut problem. ($M(n, m, U)$ is the time needed for solving the maximum flow problem on a network with n nodes, m arcs, and U as the largest arc capacity.) 2. Determines minimum cuts between all pairs of nodes in the network by solving $(n - 1)$ maximum flow problems. 3. Can be used to determine the minimum value cuts between all pairs of nodes in the case in which we define the value of a cut differently than the capacity of the cut. 4. The $O(n^3)$ term in the worst-case bound can be reduced to $O(n^2)$ using different data structures. 5. Applicable to undirected networks only.

Figure 8.19 *(Continued)*

shows that by exploiting specialized structures or advanced data structures, we can improve on the running time of maximum flow computations, sometimes dramatically.

REFERENCE NOTES

We present the reference notes in this chapter separately for each of the several topics related to maximum flows that we have studied in this chapter.

Flows in unit capacity networks. Even and Tarjan [1975] showed that Dinic's algorithm solves the maximum flow problem in unit capacity and unit capacity simple networks in $O(\min\{n^{2/3}m, m^{3/2}\})$ and $O(n^{1/2}m)$ time, respectively. The algorithms we presented in Section 8.2 are due to Ahuja and Orlin [1991]; they use similar ideas and have the same running times. Fernandez-Baca and Martel [1989] presented and analyzed algorithms for solving more general maximum flow problems with "small" integer capacities.

Flows in bipartite networks. By improving on the running times of Dinic's [1970] and Karzanov's [1974] algorithms, Gusfield, Martel, and Fernandez-Baca [1987] developed the first specializations of maximum flow algorithms for bipartite networks. Ahuja, Orlin, Stein, and Tarjan [1990] provided further improvements and showed that it is possible to substitute n_1 for n in the time bounds of almost all preflow-push algorithms to obtain new time bounds for bipartite networks (recall that n_1 is the number of nodes on the smaller side of the bipartite network). This result implies that the generic preflow-push algorithm, the FIFO implementation, the highest-label implementation, and the excess scaling algorithm can solve the maximum flow problem in bipartite networks in $O(n_1^2m)$, $O(n_1m + n_1^3)$, $O(n_1m + n_1^2\sqrt{m})$, and $O(n_1m + n_1^2 \log U)$ time. Our discussion of the bipartite preflow-push algorithm in Section 8.3 is adapted from this paper. We have adapted the baseball elimination application from Schwartz [1966], and the network reliability application from Van Slyke and Frank [1972]. The paper by Gusfield, Martel, and Fernandez-Baca [1987] describes additional applications of bipartite maximum flow problems.

Flows in planar networks. In Section 8.4 we discussed the relationship between minimum s–t cuts in a network and shortest paths in its dual. Given a planar network G, the algorithm of Hopcroft and Tarjan [1974] constructs a planar representation in $O(n)$ time; from this representation, we can construct the dual network in $O(n)$ time. Berge [1957] showed that augmenting flow along certain paths, called *superior paths*, provides an algorithm that finds a maximum flow within n augmentations. Itai and Shiloach [1979] described an $O(n \log n)$ implementation of this algorithm. Hassin [1981] showed how to compute a maximum flow from the shortest path distances in the dual network. We have presented this method in our discussion in Section 8.4. For faster maximum flow algorithms in planar (but not necessarily s–t planar) undirected and directed networks, see Johnson and Venkatesan [1982] and Hassin and Johnson [1985].

Dynamic tree implementation. Sleator and Tarjan [1983] developed the dynamic tree data structure and used it to improve the worst-case complexity of Dinic's algorithm from $O(n^2m)$ to $O(nm \log n)$. Since then, researchers have used this data structure on many occasions to improve the performance of a range of network flow algorithms. Using the dynamic tree data structure, Goldberg and Tarjan [1986] improved the complexity of the FIFO preflow-push algorithm (described in Section 7.7) from $O(n^3)$ to $O(nm \log (n^2/m))$, and Ahuja, Orlin, and Tarjan [1989] improved the complexity of the excess scaling algorithm (described in Section 7.9) and several of its variants.

Network connectivity. Even and Tarjan [1975] offered an early discussion of arc connectivity of networks. Some of our discussion in Section 8.6 uses their results. The book by Even [1979] also contains a good discussion on node connectivity of a network. The $O(nm)$ time arc connectivity algorithm (for undirected networks) that we presented in Section 8.6 is due to Matula [1987] and is currently the fastest available algorithm. Mansour and Schieber [1988] presented an $O(nm)$ algorithm for determining the arc connectivity of a directed network.

All-pairs minimum value cut problem. Gomory and Hu [1961] developed the first algorithm for solving the all-pairs minimum cut problem on undirected networks that solves a sequence of $(n - 1)$ maximum flow problems. Gusfield [1990] presented an alternate all-pairs minimum cut algorithm that is very easy to implement using a code for the maximum flow problem. Talluri [1991] described yet a third approach. The algorithm we described in Section 8.7, which is due to Cheng and Hu [1990], is more general since it can handle cases when the value of a cut is defined differently than its capacity. Unfortunately, no one yet knows how to solve the all-pairs minimum value cut problem in directed networks as efficiently. No available algorithm is more efficient than solving $\Omega(n^2)$ maximum flow problems. The application of the all-pairs minimum value cut problem that we described at the end of Section 8.7 is due to Gusfield [1988]. Hu [1974] describes an additional application of the all-pairs minimum value cut problem that arises in network design.

EXERCISES

8.1 (a) Show that it is always possible to decompose a circulation in a unit capacity network into unit flows along arc-disjoint directed cycles.
(b) Show that it is always possible to decompose a circulation in a simple network into unit flows along node-disjoint directed cycles.

8.2. Let $G = (N, A)$ be a directed network. Show that it is possible to decompose the arc set A into an arc-disjoint union of directed cycles if and only if G has a circulation x with $x_{ij} = 1$ for every arc $(i, j) \in A$. Moreover, show that we can find such a solution if and only if the indegree of each node equals its outdegree.

8.3. An undirected network is *biconnected* if it contains two node disjoint paths between every pair of nodes (except, of course, at the starting and terminal points). Show that a biconnected network must satisfy the following three properties: (1) for every two nodes p and q, and any arc (k, l), some path from p to q contains arc (k, l); (2) for every three nodes p, q, and r, some path from p to r contains node q; (3) for every three nodes p, q, and r, some path from p to r does not contain q.

8.4. Suppose that you are given a maximum flow problem in which all arc capacities are the same. What is the most efficient method (from the worst-case complexity point of view) to solve this problem?

8.5. Using the unit capacity maximum flow algorithm, establish a maximum flow in the network shown in Figure 8.20.

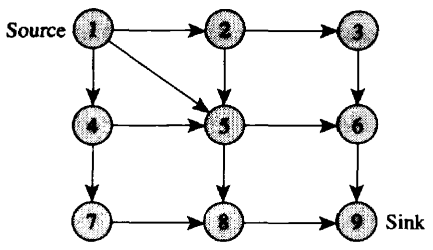


Figure 8.20 Example for Exercise 8.5.

8.6. Adapt the unit capacity maximum flow algorithm for unit capacity simple bipartite networks. In doing so, try to obtain the best possible running time. Describe your algorithm and analyze its worst-case complexity.

8.7. Consider a generalization of unit capacity networks in which arcs incident to the source and the sink nodes can have arbitrary capacities, but the remainder of the arcs have unit capacities. Will the unit capacity maximum flow algorithm still solve the problem in $O(\min\{n^{2/3}m, m^{3/2}\})$ time, or might the algorithm require more time? Consider a further generalization of the problem in which arcs incident to the source, the sink, and one other node have arbitrary capacities. What will be the complexity of the unit capacity maximum flow algorithm when applied to this problem?

8.8. We define a class of networks to be *small-capacity networks* if each arc capacity is between 1 and 4. Describe a generalization of the unit capacity maximum flow algorithm that would solve the maximum flow problems on small-capacity networks in $O(\min\{n^{2/3} m, m^{3/2}\})$ time.

8.9. What is the best possible bound you can obtain on the running time of the generic preflow-push algorithm applied to unit capacity networks?

8.10. Suppose we apply the preflow-push algorithm on a unit capacity simple network with the modification that we do not perform push/relabel operations on any node whose distance label exceeds $n^{1/2}$.
(a) Show that the modified preflow-push algorithm terminates within $O(n^{1/2}m)$ time.
(b) Show that at the termination of the algorithm, the maximum additional flow that can reach the sink is at most $n^{1/2}$.
(c) Can you convert this preflow into a maximum flow in $O(n^{1/2}m)$ time? If yes, then how?

8.11. Let x be a flow in a directed network. Assume that x is not a maximum flow. Let P and P' denote two successive shortest paths (i.e., P' is the shortest path after augmentation on path P) and suppose that P' contains at least one arc whose reversal lies in P. Show that $|P'| \geq |P| + 2$.

8.12. This exercise concerns the baseball elimination problem discussed in Application 8.1. Show that we can eliminate team 0 if and only if some nonempty set S of nodes satisfies the condition that

$$w_{\max} < \frac{\sum_{i \in S} w_i + \sum_{1 \leq i < j \leq n} g_{ij} - \sum_{i \notin s \text{ and } j \notin s} g_{ij}}{|S|}.$$

(*Hint*: Use the max-flow min-cut theorem.)

8.13. Given two n-element arrays α and β, we want to know whether we can construct an n-node directed graph so that node i has outdegree equal to $\alpha(i)$ and an indegree equal to $\beta(i)$. Show how to solve this problem by solving a maximum flow problem. [*Hint*: Transform this problem to a feasible flow problem, as described in Application 6.1, on a complete bipartite graph $G = (N_1 \cup N_2, A)$ with $N_1 = \{1, 2, \ldots, n\}$, $N_2 = \{1', 2', \ldots, n'\}$, and $A = N_1 \times N_2$.]

8.14. Given an n-node graph G and two n-element arrays α and β, we wish to determine whether some subgraph G' of G satisfies the property that for each node i, $\alpha(i)$ and $\beta(i)$ are the outdegree and indegree of node i. Formulate this problem as a maximum flow problem. (*Hint*: The transformation is similar to the one used in Exercise 8.13.)

8.15. Apply the bipartite preflow-push algorithm to the maximum flow problem given in Figure 8.21. Among all the active nodes, push flow from a node with the smallest distance label.

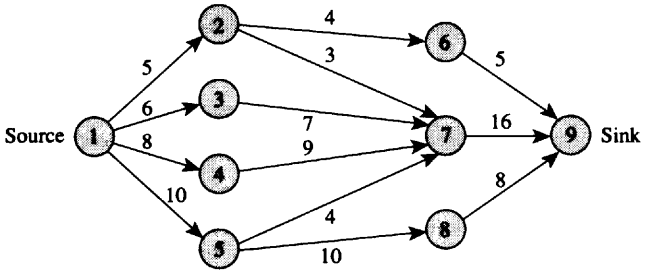


Figure 8.21 Example for Exercise 8.15.

8.16. Consider a bipartite network $G = (N_1 \cup N_2, A)$ with $n_1 = |N_1| \le |N_2| = n_2$. Show that when applied to this network, the shortest augmenting path algorithm performs $O(n_1(n_1 + n_2)) = O(n_1 n_2)$ relabel steps and $O(n_1 m)$ augmentations, and runs in $O(n_1^2 m)$ time.

8.17. Suppose that we wish to find the maximum flow between two nodes in a bipartite network $G = (N_1 \cup N_2, A)$ with $n_1 = |N_1|$ and $n_2 = |N_2|$. This exercise considers the development of faster special implementations of the generic bipartite preflow-push algorithms.

(a) Show that if the algorithm always pushes flow from an active node with the highest distance label, it runs in $O(n_1^3 + n_1 m)$ time.

(b) Show that if the algorithm examines active nodes in a FIFO order, it runs in $O(n_1^3 + n_1 m)$ time.

(c) Develop an excess scaling version of the generic bipartite flow algorithm and show that it runs in $O(n_1 m + n_1^2 \log U)$ time.

8.18. A *semi-bipartite network* is defined as a network $G = (N, A)$ whose node set N can be partitioned into two subsets N_1 and N_2 so that no arc has both of its endpoints in N_2 (i.e., we allow arcs with both of their endpoints in N_1). Let $n_1 = |N_1|$, $n_2 = |N_2|$, and $n_1 \le n_2$. Show how to modify the generic bipartite preflow-push algorithm so that it solves the maximum flow problem on semi-bipartite networks in $O(n_1^2 m)$ time.

8.19. (a) Prove that the graph shown in Figure 8.6(a) cannot be planar. (*Hint*: Use Euler's formula.)

(b) Prove that the graph shown in Figure 8.6(b) cannot be planar. (*Hint*: Use Euler's formula.)

8.20. Show that an undirected planar network always contains a node with degree at most 5.

8.21. Apply the planar maximum flow algorithm to identify a maximum flow in the network shown in Figure 8.22.

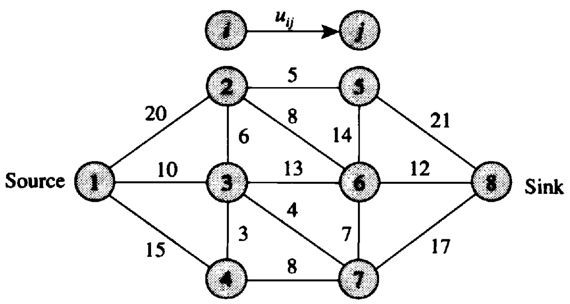


Figure 8.22 Example for Exercise 8.21.

8.22. Duals of directed *s–t* planar networks. We define the dual graph of a directed s–t planar network $G = (N, A)$ as follows. We first draw an arc (t, s) of zero capacity, which divides the unbounded face into two faces: a new unbounded face and a new bounded face. We then place a node f^* inside each face f of the primal network G. Let s^* and t^*, respectively, denote the nodes in the dual network corresponding to the new bounded face and the new unbounded face. Each arc $(i, j) \in A$ lies on the boundary of the two faces f_1 and f_2; corresponding to this arc, the dual graph contains two oppositely directed arcs (f_1, f_2) and (f_2, f_1). If arc (i, j) is a clockwise arc in the face f_1, we define the cost of arc (f_1, f_2) as u_{ij} and the cost of (f_2, f_1) as zero. We define arc costs in the opposite manner if arc (i, j) is a counterclockwise arc in the face f_1. Construct the dual of the s–t planar network shown in Figure 8.20. Next show that there is a one-to-one correspondence between s–t cuts in the primal network and directed paths from node s^* to node t^* in the dual network; moreover, show that the capacity of the cut equals the cost of the corresponding path.

8.23. Show that if G is an s–t planar directed network, the minimum number of arcs in a directed path from s to t is equal to the maximum number of arc-disjoint s–t cuts. (*Hint*: Apply Theorem 6.7 to the dual of G.)

8.24. Node coloring algorithm. In *the node coloring problem*, we wish to color the nodes of a network so that the endpoints of each arc have a different color. In this exercise we discuss an algorithm for coloring a planar undirected graph using at most six colors. The algorithm first orders the nodes of the network using the following iterative loop: It selects a node with degree at most 5 (from Exercise 8.20, we can always find such a node), deletes this node and its incident arcs from the network, and updates the degrees of all the nodes affected. The algorithm then examines nodes in the reverse order and assigns colors to them.

(a) Explain how to assign colors to the nodes to create a valid 6-coloring (i.e., the endpoints of every arc have a different color). Justify your method.

(b) Show how to implement the node coloring algorithm so that it runs in $O(m)$ time.

8.25. Consider the numerical example used in Section 8.5 to illustrate the dynamic tree implementation of the shortest augmenting path algorithm. Perform further iterations of the algorithm starting from Figure 8.11(c) until you find a maximum flow that has a value of 3.

8.26. Solve the maximum flow problem given in Figure 8.23 by the dynamic tree implementation of the shortest augmenting path algorithm.

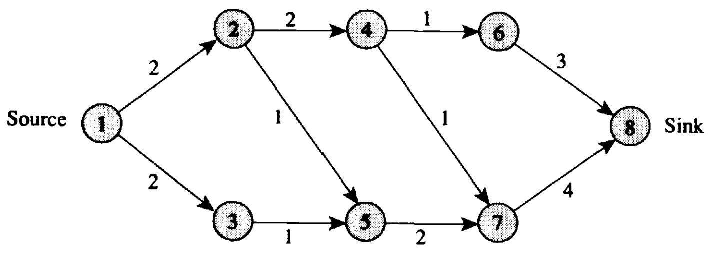


Figure 8.23 Example for Exercise 8.26.

8.27. Show how to use the dynamic tree data structure to implement in $O(m \log n)$ time the algorithm described in Exercise 7.11 for converting a maximum preflow into a maximum flow.

8.28. In Section 3.5 we showed how we can determine the flow decomposition of any flow in $O(nm)$ time. Show how to use the dynamic tree data structure to determine the flow decomposition in $O(m \log n)$ time.

8.29. Prove Property 8.11.

8.30. Compute the arc connectivity for the networks shown in Figure 8.24. Feel free to determine the maximum number of arc-disjoint paths between any pairs of nodes by inspection.

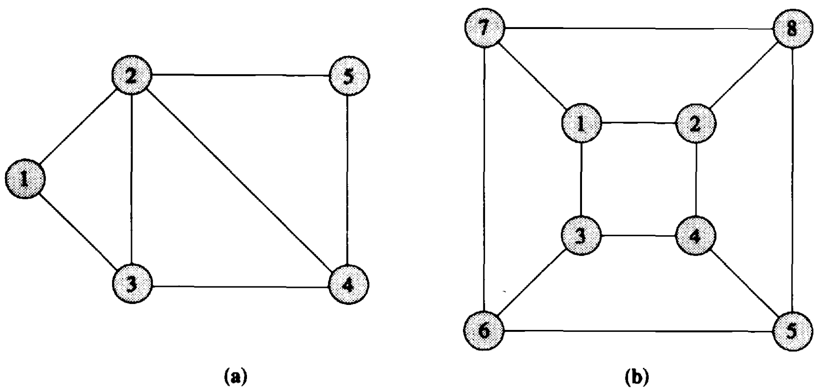


Figure 8.24 Example for Exercises 8.30 and 8.36.

8.31. Construct an undirected network whose nodes all have degree at least 3, but whose arc connectivity is 2.

8.32. An undirected network is said to be *k-connected* if every pair of nodes are connected by at least k arc-disjoint paths. Describe an $O(m)$ algorithm for determining whether a network is 1-connected. Use this algorithm to describe a simple $O(knm)$ algorithm for determining whether a network is k-connected. This algorithm should be different than those described in Section 8.6.

8.33. In a directed graph G, we define the *arc connectivity*, $\alpha[i, j]$, of an ordered pair $[i, j]$ of nodes as the minimum number of arcs whose deletion from the network eliminates all the directed path from node i to node j. We define the arc connectivity of a network G as $\alpha(G) = \min\{\alpha[i, j] : [i, j] \in N \times N\}$. Describe an algorithm for determining $\alpha(G)$ in $O(\min\{n^{5/3}m, m^2\})$ time and prove that this algorithm correctly determines the arc connectivity of any directed network. (*Hint*: Let p be a node in G of minimum degree. Determine $\alpha[p, j]$ and $\alpha[j, p]$ for all j, and take the minimum of these numbers.)

8.34. Arc connectivity of directed networks (Schnorr [1979]). In Exercise 8.33 we showed how to determine the arc connectivity $\alpha(G)$ of a directed network G by solving at most $2n$ maximum flow problems. Prove the following result, which would enable us to determine the arc connectivity by solving n maximum flow problems. Let $1, 2, \ldots, n$ be any ordering of the nodes in the network, and let node $(n + 1) = 1$. Show that $\alpha(G) = \min\{\alpha[i, i + 1]: i = 1, \ldots, n\}$.

8.35. Node connectivity of undirected networks. We define the ***node connectivity***, $\beta[i, j]$, of a pair $[i, j]$ of nodes in an undirected graph $G = (N, A)$ as the minimum number of nodes whose deletion from the network eliminates all directed paths from node i to node j.

(a) Show that if $(i, j) \in A$, then $\beta[i, j]$ is not defined.

(b) Let $H = \{[i, j] \in N \times N: (i, j) \notin A\}$ and let $\beta(G) = \min\{\beta[i, j]: [i, j] \in H\}$ denote the node connectivity of a network G. Show that $\beta(G) \le 2\lfloor m/n \rfloor$.

(c) Show that the node connectivity of a network is no more than its arc connectivity.

(d) A natural strategy for determining the node connectivity of a network would be to generalize an arc connectivity algorithm described in Section 8.6. We fix a node p and determine $\beta[p, j]$ for each j for which $(p, j) \notin A$. Using Figure 8.25 show that the minimum of these values will not be equal to $\beta(G)$. Explain why this approach fails for finding node connectivity even though it works for finding arc connectivity.

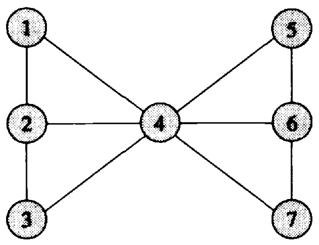


Figure 8.25 Example for Exercise 8.35.

8.36. Solve the all-pairs minimum cut problems given in Figure 8.24. Obtain the separator tree for each problem.