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
6.2 Applications
6.3 Flows and Cuts
6.4 Generic Augmenting Path Algorithm
6.5 Labeling Algorithm and the Max-Flow Min-Cut Theorem
6.6 Combinatorial Implications of the Max-Flow Min-Cut Theorem
6.7 Flows with Lower Bounds
6.8 Summary

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.

166

Ahuja, et. al. Network Flows
©1993 Prentice-Hall
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 \( U = \max\{u_{ij} : (i, j) \in A\} \) and \( (i, j) \in A \). 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.

Maximize \( v \) 
subject to

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

\( 0 \leq x_{ij} \leq u_{ij} \) for each \( (i, j) \in A \).

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:

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

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

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.

![Diagram of the network representing the problem of representatives.](image)

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 $\alpha_i$ and column sums $\beta_j$. 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 by solving a feasible flow problem for a network with nonnegative lower bounds on arc flows. (As shown in Sectin 6.7, we can solve this problem by solving a maximum flow problem 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

<table>
<thead>
<tr>
<th>Row sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
</tr>
<tr>
<td>9.6</td>
</tr>
<tr>
<td>3.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.3</td>
</tr>
</tbody>
</table>

Figure 6.2 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 \([d_{ij}]\) and \([d_{ij}]\), 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 \(\alpha\) as the greatest multiple of \(k\) less than or equal to \(\alpha\) and the smallest multiple of \(k\) greater than or equal to \(\alpha\).

**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 \( P = 2 \mid J \mid - 1 \) mutually disjoint intervals of dates between consecutive milestones. Let \( T_{k,t} \) 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.

<table>
<thead>
<tr>
<th>Job ((j))</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing time ((p_j))</td>
<td>1.5</td>
<td>1.25</td>
<td>2.1</td>
<td>3.6</td>
</tr>
<tr>
<td>Release time ((r_j))</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Due date ((d_j))</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

**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 \( T_{k,t} \), 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 \( T_{k,t} \) 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 \( T_{k,t} \) if \( r_j \leq k \) and \( d_j \geq l + 1 \) by an arc with capacity \((l - k + 1)M\) 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.

Sec. 6.2 Applications
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 $\alpha_i$ and $\beta_i$ 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 $\beta_i$ and an arc $(i, t)$ of capacity $\alpha_i$. 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.
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 $\sum_{i \in A_1} \alpha_i + \sum_{i \in A_2} \beta_i + \sum_{(i,j) \in A_1 \times A_2} c_{ij}$. 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 $\alpha_i$, an arc $(s, i)$ for every $i \in A_2$ of capacity $\beta_i$, 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.

![Shipment Table](image)

<table>
<thead>
<tr>
<th>Shipment</th>
<th>Origin</th>
<th>Destination</th>
<th>Delivery Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Port A</td>
<td>Port C</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Port A</td>
<td>Port C</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Port B</td>
<td>Port D</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Port B</td>
<td>Port C</td>
<td>6</td>
</tr>
</tbody>
</table>

![Transit Times Matrix](image)

**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),
... Consequently, the cost assignment of the goods between several plants, the customers have demands must reach their respective steamship company on or before the delivery dates... Consider an example characteristics shown in Figure 6.9(a). The company (or C) on day 3. 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 sequences of two consecutive shipments; 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 will later 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 \( (i, j) \) \( \in \mathcal{A} \) 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
arcs whose endpoints belong to the different subsets $S$ and $\overline{S}$. We refer to a cut as an $s$–$t$ cut if $s \in S$ and $t \in \overline{S}$. We also refer to an arc $(i, j)$ with $i \in S$ and $j \in \overline{S}$ as a forward arc of the cut, and an arc $(i, j)$ with $i \in \overline{S}$ and $j \in S$ as a backward arc of the cut $[S, \overline{S}]$. Let $(S, \overline{S})$ denote the set of forward arcs in the cut, and let $(\overline{S}, S)$ denote the set of backward arcs. For example, in Figure 6.11, the dashed arcs constitute an $s$–$t$ cut. For this cut, $(S, \overline{S}) = \{(1, 2), (3, 4), (5, 6)\}$, and $(\overline{S}, S) = \{(2, 3), (4, 5)\}$.

**Figure 6.11** Example of an $s$–$t$ cut.

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

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

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 $\overline{S}$ 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 $r[S, \overline{S}]$ of an $s$–$t$ cut $[S, \overline{S}]$ as the sum of the residual capacities of forward arcs in the cut. That is,
\[ r[S, \overline{S}] = \sum_{(i,j) \in (S, \overline{S})} r_{ij}. \]

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

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 \( S \), then \( x_{pq} \) does not appear in the expression. This observation implies that
\[
v = \sum_{(i,j) \in (S, \overline{S})} x_{ij} - \sum_{(i,j) \in (S,S)} x_{ij}, \quad (6.3)
\]

The first expression on the right-hand side of (6.3) denotes the amount of flow from the nodes in \( S \) to nodes in \( \overline{S} \), and the second expression denotes the amount of flow returning from the nodes in \( \overline{S} \) 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 \([S, \overline{S}]\) equals \( v \). Substituting \( x_{ij} \leq u_{ij} \) in the first expression of (6.3) and \( x_{ij} \geq 0 \) in the second expression shows that
\[
v \leq \sum_{(i,j) \in (S, \overline{S})} u_{ij} = u[S, \overline{S}], \quad (6.4)
\]

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 \([S, \overline{S}]\), then \( x \) is a maximum flow and the cut \([S, \overline{S}]\) is a minimum cut. The max-flow min-cut theorem, proved in Section 6-5, 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 \( \Delta v \geq 0 \). The inequality (6.4) implies that
\[
v + \Delta v \leq \sum_{(i,j) \in (S, \overline{S})} u_{ij}, \quad (6.5)
\]

Subtracting (6.3) from (6.5) shows that
\[
\Delta v \leq \sum_{(i,j) \in (S, \overline{S})} (u_{ij} - x_{ij}) + \sum_{(i,j) \in (S,S)} x_{ij}. \quad (6.6)
\]
We now use Assumption 6.4 to note that we can rewrite $\sum_{(i,j) \in (S, \bar{S})} x_{ij}$ as $\sum_{(i,j) \in (S, \bar{S})} x_{ij}$. Consequently,
\[
\Delta u \leq \sum_{(i,j) \in (S, \bar{S})} (u_{ij} - x_{ij} + x_{ij}) = \sum_{(S, \bar{S})} r_{ij}.
\]
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 $\delta = \min\{r_{13}, r_{22}, r_{24}\} = \min\{1, 2, 1\} = 1$. Observe that, by definition, the capacity $\delta$ 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
\( \phi \in \mathbb{R} \) \( x_{ij} \) as

\[ \sum_{(i,j) \in E} x_{ij} \]

\( \text{considering path} \)

\( \text{equal to the} \)

\( \text{algorithm for} \)

\( \text{augment path} \)

\( \text{to the residual network} \)

\( \text{augment path as} \), the residual

\( \text{path} = 1 \). Observe

\( \text{positive. Con-}

\( \text{by identifying work contains algorithm is}

\[ \text{Figure 6.13 Illustrating the generic augmenting path algorithm: (a) residual network}
\]

\[ \text{for the zero flow; (b) network after augmenting four units along the path 1-3-4; (c) network}
\]

\[ \text{after augmenting one unit along the path 1-2-4-3; (d) network after augmenting one unit along the path 1-2-4.} \]

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

\[ \text{Relationship between the Original and Residual Networks} \]

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

Sec. 6.4 Generic Augmenting Path Algorithm
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 \( \delta \) units on arc \((i, j)\) in the residual network corresponds to (1) an increase in \( x_{ij} \) by \( \delta \) units in the original network, or (2) a decrease in \( x_{ji} \) by \( \delta \) 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
one such choice as follows. To highlight this choice, let us rewrite \( r_{ij} = u_{ij} - x_{ij} + x_{ji} = u_{ij} - r_{ij} \) as \( x_{ij} = u_{ij} - r_{ij} \) 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](image)

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](image)

Figure 6.16 The effect of augmentation on flow decomposition: (a) the two augmentations \( P_1P_2P_3 \) and \( Q_1Q_2Q_3 \); (b) net effect of these augmentations.

**Sec. 6.4 Generic Augmenting Path Algorithm**
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 \( \overline{S} = N - S \) is the set of unlabeled nodes. Clearly, \( s \in S \) and \( t \in \overline{S} \). Since the algorithm cannot label any node in \( \overline{S} \) from any node in \( S \), \( r_{ij} = 0 \) for each \( (i, j) \in (S, \overline{S}) \). Furthermore, since \( r_{ij} = (u_{ij} - x_{ij}) + x_{ji} \), \( x_{ij} \equiv u_{ij} \) and \( x_{ji} \geq 0 \), the condition \( r_{ij} = 0 \) implies that \( x_{ij} = u_{ij} \) for every arc \( (i, j) \in (S, \overline{S}) \) and \( x_{ji} = 0 \) for every arc \( (i, j) \in (\overline{S}, S) \). [Recall our assumption that for each arc \( (i, j) \in A, \)

184

Maximum Flows: Basic Ideas  Chap. 6
flow decomposition. In our some immensity that the ates in finite a maximum tation of the The labeling on the ideas ations of this section 3.4 to them fans out urce along a ritioned the d nodes are s and so the nodes in the thm has not cts a labeled ch and label um sends the es the labels ll the labeled pt connected ; description

\begin{algorithm} \textbf{labeling;} \\
\textbf{begin} \\
\quad \text{label node } t; \\
\quad \text{while } t \text{ is labeled do} \\
\qquad \text{begin} \\
\qquad \quad \text{unlabel all nodes;} \\
\qquad \quad \text{set } \text{pred}(j) = 0 \text{ for each } j \in N; \\
\qquad \quad \text{label node } s \text{ and set } \text{LIST} = \{s\}; \\
\qquad \quad \text{while } \text{LIST} \neq \emptyset \text{ and } t \text{ is unlabeled do} \\
\qquad \qquad \text{begin} \\
\qquad \qquad \quad \text{remove a node } i \text{ from } \text{LIST}; \\
\qquad \qquad \quad \text{for each arc } (i, j) \text{ in the residual network emanating from node } i \text{ do} \\
\qquad \qquad \qquad \quad \text{if node } j \text{ is unlabeled then set } \text{pred}(j) = i, \text{ label node } j, \text{ and} \\
\qquad \qquad \qquad \quad \quad \text{add } j \text{ to LIST}; \\
\qquad \qquad \quad \text{end;} \\
\qquad \quad \text{if } t \text{ is labeled then } \text{augment} \\
\qquad \text{end;} \\
\text{end;} \\
\text{procedure } \text{augment;} \\
\text{begin} \\
\quad \text{use the predecessor labels to trace back from the sink to the source to} \\
\quad \text{obtain an augmenting path } P \text{ from node } s \text{ to node } t; \\
\quad \delta = \min \{r_i : (i, j) \in P\}; \\
\quad \text{augment } \delta \text{ units of flow along } P \text{ and update the residual capacities;} \\
\text{end;} \\
\end{algorithm}

\textbf{Figure 6.17} Labeling algorithm.

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

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

This discussion shows that the value of the current flow \(x\) equals the capacity of the cut \([S, \overline{S}]\). But then Property 6.1 implies that \(x\) is a maximum flow and \([S, \overline{S}]\) 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.

\textbf{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.

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

\textbf{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
\[ s-t \text{ cut } (S, \overline{S}) \text{ whose capacity equals the flow value, thereby implying that the flow must be maximum.} \]

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 $10^8$ times, each time augmenting unit flow along the path. This example illustrates one shortcoming of the algorithm.

![Pathological example of the labeling algorithm](image)

**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.

Sec. 6.5 **Labeling Algorithm and 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 $[S, \overline{S}]$. 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 node $s$ 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 $\infty$ 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 \( \infty \).

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 forward arcs in the cut \( Q \) must be artificial arcs because the original arcs have infinite capacity. From \( Q \) we construct a set \( H \) of nodes as follows: if \((s, i) \in Q \) or \((i, t) \in Q \), 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 \((5, 5')\).

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

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

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

Sec. 6.7 Flows with Lower Bounds
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 Groeneveld [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]).

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 Queridanne [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 $i$th family has $a(i)$ members. Also assume that $q$ tables are available and that the $j$th 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:

<table>
<thead>
<tr>
<th>Department</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(6, 8)</td>
<td>(11, 12)</td>
<td>(7, 12)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(4, 6)</td>
<td>(11, 12)</td>
<td>(7, 12)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2, 4)</td>
<td>(10, 12)</td>
<td>(5, 7)</td>
</tr>
</tbody>
</table>