Chapter 7

Network Flow

In this chapter, we focus on a rich set of algorithmic problems that grow, in a sense, out of one of the original problems we formulated at the beginning of the course: *Bipartite Matching*.

Recall the set-up of the Bipartite Matching Problem. A *bipartite graph* $G = (V, E)$ is an undirected graph whose node set can be partitioned as $V = X \cup Y$, with the property that every edge $e \in E$ has one end in X and the other end in Y. We often draw bipartite graphs as in Figure 7.1, with the nodes in X in a column on the left, the nodes in Y in a column on the right, and each edge crossing from the left column to the right column.

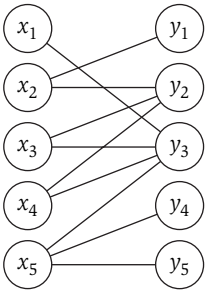

Figure 7.1 A bipartite graph.

Now, we've already seen the notion of a *matching* at several points in the course: We've used the term to describe collections of pairs over a set, with the property that no element of the set appears in more than one pair. (Think of men (X) matched to women (Y) in the Stable Matching Problem, or characters in the Sequence Alignment Problem.) In the case of a graph, the edges constitute pairs of nodes, and we consequently say that a *matching* in a graph $G = (V, E)$ is a set of edges $M \subseteq E$ with the property that each node appears in at most one edge of M. A set of edges M is a *perfect matching* if every node appears in exactly one edge of M.

Matchings in bipartite graphs can model situations in which objects are being *assigned* to other objects. We have seen a number of such situations in our earlier discussions of graphs and bipartite graphs. One natural example arises when the nodes in X represent jobs, the nodes in Y represent machines, and an edge (x_i, y_j) indicates that machine y_j is capable of processing job x_i. A perfect matching is, then, a way of assigning each job to a machine that can process it, with the property that each machine is assigned exactly one job. Bipartite graphs can represent many other relations that arise between two

distinct sets of objects, such as the relation between customers and stores; or houses and nearby fire stations; and so forth.

One of the oldest problems in combinatorial algorithms is that of determining the size of the largest matching in a bipartite graph G. (As a special case, note that G has a perfect matching if and only if $|X| = |Y|$ and it has a matching of size $|X|$.) This problem turns out to be solvable by an algorithm that runs in polynomial time, but the development of this algorithm needs ideas fundamentally different from the techniques that we've seen so far.

Rather than developing the algorithm directly, we begin by formulating a general class of problems—*network flow* problems—that includes the Bipartite Matching Problem as a special case. We then develop a polynomial-time algorithm for a general problem, the *Maximum-Flow Problem*, and show how this provides an efficient algorithm for Bipartite Matching as well. While the initial motivation for network flow problems comes from the issue of traffic in a network, we will see that they have applications in a surprisingly diverse set of areas and lead to efficient algorithms not just for Bipartite Matching, but for a host of other problems as well.

7.1 The Maximum-Flow Problem and the Ford-Fulkerson Algorithm

The Problem

One often uses graphs to model *transportation networks*—networks whose edges carry some sort of traffic and whose nodes act as "switches" passing traffic between different edges. Consider, for example, a highway system in which the edges are highways and the nodes are interchanges; or a computer network in which the edges are links that can carry packets and the nodes are switches; or a fluid network in which edges are pipes that carry liquid, and the nodes are junctures where pipes are plugged together. Network models of this type have several ingredients: *capacities* on the edges, indicating how much they can carry; *source* nodes in the graph, which generate traffic; *sink* (or destination) nodes in the graph, which can "absorb" traffic as it arrives; and finally, the traffic itself, which is transmitted across the edges.

Flow Networks We'll be considering graphs of this form, and we refer to the traffic as *flow*—an abstract entity that is generated at source nodes, transmitted across edges, and absorbed at sink nodes. Formally, we'll say that a *flow network* is a directed graph $G = (V, E)$ with the following features.

- Associated with each edge e is a *capacity*, which is a nonnegative number that we denote c_e.

- There is a single *source* node $s \in V$.
- There is a single *sink* node $t \in V$.

Nodes other than s and t will be called *internal* nodes.

We will make two assumptions about the flow networks we deal with: first, that no edge enters the source s and no edge leaves the sink t; second, that there is at least one edge incident to each node; and third, that all capacities are integers. These assumptions make things cleaner to think about, and while they eliminate a few pathologies, they preserve essentially all the issues we want to think about.

Figure 7.2 illustrates a flow network with four nodes and five edges, and capacity values given next to each edge.

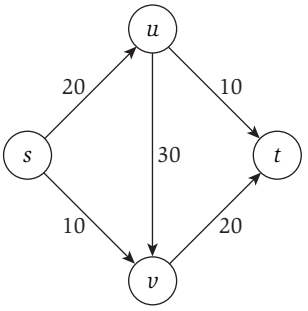

Figure 7.2 A flow network, with source s and sink t. The numbers next to the edges are the capacities.

Defining Flow Next we define what it means for our network to carry traffic, or flow. We say that an *s-t flow* is a function f that maps each edge e to a nonnegative real number, $f : E \to \mathbf{R}^+$; the value $f(e)$ intuitively represents the amount of flow carried by edge e. A flow f must satisfy the following two properties.[1]

(i) *(Capacity conditions)* For each $e \in E$, we have $0 \leq f(e) \leq c_e$.

(ii) *(Conservation conditions)* For each node v other than s and t, we have

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e).$$

Here $\sum_{e \text{ into } v} f(e)$ sums the flow value $f(e)$ over all edges entering node v, while $\sum_{e \text{ out of } v} f(e)$ is the sum of flow values over all edges leaving node v.

Thus the flow on an edge cannot exceed the capacity of the edge. For every node other than the source and the sink, the amount of flow entering must equal the amount of flow leaving. The source has no entering edges (by our assumption), but it is allowed to have flow going out; in other words, it can generate flow. Symmetrically, the sink is allowed to have flow coming in, even though it has no edges leaving it. The *value* of a flow f, denoted $v(f)$, is defined to be the amount of flow generated at the source:

$$v(f) = \sum_{e \text{ out of } s} f(e).$$

To make the notation more compact, we define $f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$ and $f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$. We can extend this to sets of vertices; if $S \subseteq V$, we

[1] Our notion of flow models traffic as it goes through the network at a steady rate. We have a single variable $f(e)$ to denote the amount of flow on edge e. We do not model *bursty* traffic, where the flow fluctuates over time.

define $f^{\text{out}}(S) = \sum_{e \text{ out of } S} f(e)$ and $f^{\text{in}}(S) = \sum_{e \text{ into } S} f(e)$. In this terminology, the conservation condition for nodes $v \neq s, t$ becomes $f^{\text{in}}(v) = f^{\text{out}}(v)$; and we can write $v(f) = f^{\text{out}}(s)$.

The Maximum-Flow Problem Given a flow network, a natural goal is to arrange the traffic so as to make as efficient use as possible of the available capacity. Thus the basic algorithmic problem we will consider is the following: Given a flow network, find a flow of maximum possible value.

As we think about designing algorithms for this problem, it's useful to consider how the structure of the flow network places upper bounds on the maximum value of an s-t flow. Here is a basic "obstacle" to the existence of large flows: Suppose we divide the nodes of the graph into two sets, A and B, so that $s \in A$ and $t \in B$. Then, intuitively, any flow that goes from s to t must cross from A into B at some point, and thereby use up some of the edge capacity from A to B. This suggests that each such "cut" of the graph puts a bound on the maximum possible flow value. The maximum-flow algorithm that we develop here will be intertwined with a proof that the maximum-flow value equals the minimum capacity of any such division, called the *minimum cut*. As a bonus, our algorithm will also compute the minimum cut. We will see that the problem of finding cuts of minimum capacity in a flow network turns out to be as valuable, from the point of view of applications, as that of finding a maximum flow.

Designing the Algorithm

Suppose we wanted to find a maximum flow in a network. How should we go about doing this? It takes some testing out to decide that an approach such as dynamic programming doesn't seem to work—at least, there is no algorithm known for the Maximum-Flow Problem that could really be viewed as naturally belonging to the dynamic programming paradigm. In the absence of other ideas, we could go back and think about simple greedy approaches, to see where they break down.

Suppose we start with zero flow: $f(e) = 0$ for all e. Clearly this respects the capacity and conservation conditions; the problem is that its value is 0. We now try to increase the value of f by "pushing" flow along a path from s to t, up to the limits imposed by the edge capacities. Thus, in Figure 7.3, we might choose the path consisting of the edges $\{(s, u), (u, v), (v, t)\}$ and increase the flow on each of these edges to 20, and leave $f(e) = 0$ for the other two. In this way, we still respect the capacity conditions—since we only set the flow as high as the edge capacities would allow—and the conservation conditions—since when we increase flow on an edge entering an internal node, we also increase it on an edge leaving the node. Now, the value of our flow is 20, and we can ask: Is this the maximum possible for the graph in the figure? If we

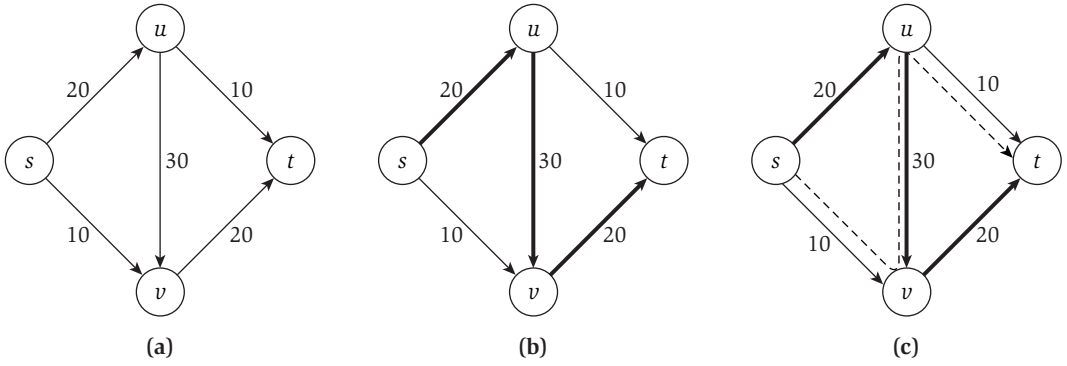

Figure 7.3 (a) The network of Figure 7.2. (b) Pushing 20 units of flow along the path s, u, v, t. (c) The new kind of augmenting path using the edge (u, v) backward.

think about it, we see that the answer is no, since it is possible to construct a flow of value 30. The problem is that we're now stuck—there is no s-t path on which we can directly push flow without exceeding some capacity—and yet we do not have a maximum flow. What we need is a more general way of pushing flow from s to t, so that in a situation such as this, we have a way to increase the value of the current flow.

Essentially, we'd like to perform the following operation denoted by a dotted line in Figure 7.3(c). We push 10 units of flow along (s, v); this now results in too much flow coming into v. So we "undo" 10 units of flow on (u, v); this restores the conservation condition at v but results in too little flow leaving u. So, finally, we push 10 units of flow along (u, t), restoring the conservation condition at u. We now have a valid flow, and its value is 30. See Figure 7.3, where the dark edges are carrying flow before the operation, and the dashed edges form the new kind of augmentation.

This is a more general way of pushing flow: We can push *forward* on edges with leftover capacity, and we can push *backward* on edges that are already carrying flow, to divert it in a different direction. We now define the *residual graph*, which provides a systematic way to search for forward-backward operations such as this.

The Residual Graph Given a flow network G, and a flow f on G, we define the *residual graph* G_f of G with respect to f as follows. (See Figure 7.4 for the residual graph of the flow on Figure 7.3 after pushing 20 units of flow along the path s, u, v, t.)

- The node set of G_f is the same as that of G.
- For each edge $e = (u, v)$ of G on which $f(e) < c_e$, there are $c_e - f(e)$ "leftover" units of capacity on which we could try pushing flow forward.

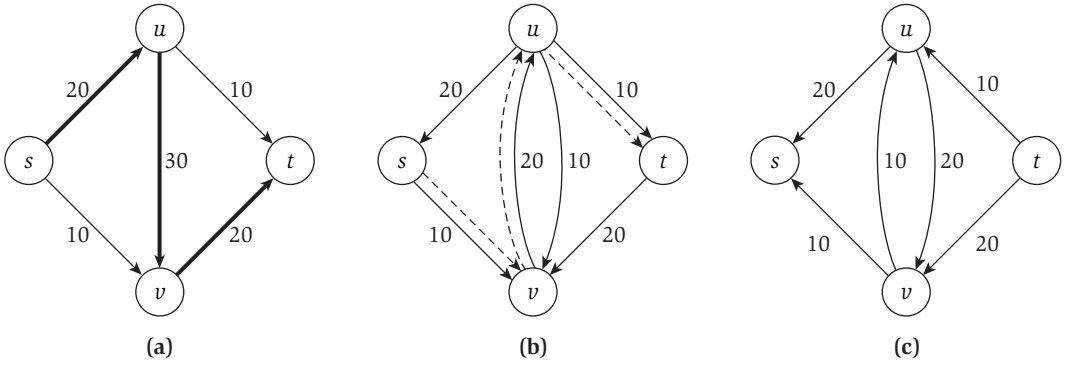

Figure 7.4 (a) The graph G with the path s, u, v, t used to push the first 20 units of flow. (b) The residual graph of the resulting flow f, with the residual capacity next to each edge. The dotted line is the new augmenting path. (c) The residual graph after pushing an additional 10 units of flow along the new augmenting path s, v, u, t.

So we include the edge $e = (u, v)$ in G_f, with a capacity of $c_e - f(e)$. We will call edges included this way *forward edges*.

- For each edge $e = (u, v)$ of G on which $f(e) > 0$, there are $f(e)$ units of flow that we can "undo" if we want to, by pushing flow backward. So we include the edge $e' = (v, u)$ in G_f, with a capacity of $f(e)$. Note that e' has the same ends as e, but its direction is reversed; we will call edges included this way *backward edges*.

This completes the definition of the residual graph G_f. Note that each edge e in G can give rise to one or two edges in G_f: If $0 < f(e) < c_e$ it results in both a forward edge and a backward edge being included in G_f. Thus G_f has at most twice as many edges as G. We will sometimes refer to the capacity of an edge in the residual graph as a *residual capacity*, to help distinguish it from the capacity of the corresponding edge in the original flow network G.

Augmenting Paths in a Residual Graph Now we want to make precise the way in which we push flow from s to t in G_f. Let P be a simple s-t path in G_f—that is, P does not visit any node more than once. We define `bottleneck`(P, f) to be the minimum residual capacity of any edge on P, with respect to the flow f. We now define the following operation `augment`(f, P), which yields a new flow f' in G.

```
augment(f, P)
  Let b = bottleneck(P, f)
  For each edge (u, v) ∈ P
    If e = (u, v) is a forward edge then
      increase f(e) in G by b
```

```
    Else ((u, v) is a backward edge, and let e = (v, u))
      decrease f(e) in G by b
    Endif
  Endfor
  Return(f)
```

It was purely to be able to perform this operation that we defined the residual graph; to reflect the importance of `augment`, one often refers to any *s*-*t* path in the residual graph as an *augmenting path*.

The result of `augment`(f, P) is a new flow f' in G, obtained by increasing and decreasing the flow values on edges of P. Let us first verify that f' is indeed a flow.

(7.1) *f' is a flow in G.*

Proof. We must verify the capacity and conservation conditions.

Since f' differs from f only on edges of P, we need to check the capacity conditions only on these edges. Thus, let (u, v) be an edge of P. Informally, the capacity condition continues to hold because if $e = (u, v)$ is a forward edge, we specifically avoided increasing the flow on e above c_e; and if (u, v) is a backward edge arising from edge $e = (v, u) \in E$, we specifically avoided decreasing the flow on e below 0. More concretely, note that `bottleneck`(P, f) is no larger than the residual capacity of (u, v). If $e = (u, v)$ is a forward edge, then its residual capacity is $c_e - f(e)$; thus we have

$$0 \leq f(e) \leq f'(e) = f(e) + \texttt{bottleneck}(P, f) \leq f(e) + (c_e - f(e)) = c_e,$$

so the capacity condition holds. If (u, v) is a backward edge arising from edge $e = (v, u) \in E$, then its residual capacity is $f(e)$, so we have

$$c_e \geq f(e) \geq f'(e) = f(e) - \texttt{bottleneck}(P, f) \geq f(e) - f(e) = 0,$$

and again the capacity condition holds.

We need to check the conservation condition at each internal node that lies on the path P. Let v be such a node; we can verify that the change in the amount of flow entering v is the same as the change in the amount of flow exiting v; since f satisfied the conservation condition at v, so must f'. Technically, there are four cases to check, depending on whether the edge of P that enters v is a forward or backward edge, and whether the edge of P that exits v is a forward or backward edge. However, each of these cases is easily worked out, and we leave them to the reader. ■

This augmentation operation captures the type of forward and backward pushing of flow that we discussed earlier. Let's now consider the following algorithm to compute an *s-t* flow in *G*.

```
Max-Flow
  Initially f(e) = 0 for all e in G
  While there is an s–t path in the residual graph G_f
    Let P be a simple s–t path in G_f
    f' = augment(f, P)
    Update f to be f'
    Update the residual graph G_f to be G_f'
  Endwhile
  Return f
```

We'll call this the *Ford-Fulkerson Algorithm*, after the two researchers who developed it in 1956. See Figure 7.4 for a run of the algorithm. The Ford-Fulkerson Algorithm is really quite simple. What is not at all clear is whether its central `While` loop terminates, and whether the flow returned is a maximum flow. The answers to both of these questions turn out to be fairly subtle.

Analyzing the Algorithm: Termination and Running Time

First we consider some properties that the algorithm maintains by induction on the number of iterations of the `While` loop, relying on our assumption that all capacities are integers.

(7.2) *At every intermediate stage of the Ford-Fulkerson Algorithm, the flow values* $\{f(e)\}$ *and the residual capacities in* G_f *are integers.*

Proof. The statement is clearly true before any iterations of the `While` loop. Now suppose it is true after j iterations. Then, since all residual capacities in G_f are integers, the value `bottleneck`(P, f) for the augmenting path found in iteration $j+1$ will be an integer. Thus the flow f' will have integer values, and hence so will the capacities of the new residual graph. ■

We can use this property to prove that the Ford-Fulkerson Algorithm terminates. As at previous points in the book we will look for a measure of *progress* that will imply termination.

First we show that the flow value strictly increases when we apply an augmentation.

(7.3) *Let* f *be a flow in* G*, and let* P *be a simple s-t path in* G_f*. Then* $v(f') = v(f) +$ `bottleneck`(P, f)*; and since* `bottleneck`$(P, f) > 0$*, we have* $v(f') > v(f)$.

Proof. The first edge e of P must be an edge out of s in the residual graph G_f; and since the path is simple, it does not visit s again. Since G has no edges entering s, the edge e must be a forward edge. We increase the flow on this edge by `bottleneck`(P, f), and we do not change the flow on any other edge incident to s. Therefore the value of f' exceeds the value of f by `bottleneck`(P, f). ■

We need one more observation to prove termination: We need to be able to bound the maximum possible flow value. Here's one upper bound: If all the edges out of s could be completely saturated with flow, the value of the flow would be $\sum_{e \text{ out of } s} c_e$. Let C denote this sum. Thus we have $v(f) \leq C$ for all s-t flows f. (C may be a huge overestimate of the maximum value of a flow in G, but it's handy for us as a finite, simply stated bound.) Using statement (7.3), we can now prove termination.

(7.4) *Suppose, as above, that all capacities in the flow network G are integers. Then the Ford-Fulkerson Algorithm terminates in at most C iterations of the* `While` *loop.*

Proof. We noted above that no flow in G can have value greater than C, due to the capacity condition on the edges leaving s. Now, by (7.3), the value of the flow maintained by the Ford-Fulkerson Algorithm increases in each iteration; so by (7.2), it increases by at least 1 in each iteration. Since it starts with the value 0, and cannot go higher than C, the `While` loop in the Ford-Fulkerson Algorithm can run for at most C iterations. ■

Next we consider the running time of the Ford-Fulkerson Algorithm. Let n denote the number of nodes in G, and m denote the number of edges in G. We have assumed that all nodes have at least one incident edge, hence $m \geq n/2$, and so we can use $O(m + n) = O(m)$ to simplify the bounds.

(7.5) *Suppose, as above, that all capacities in the flow network G are integers. Then the Ford-Fulkerson Algorithm can be implemented to run in $O(mC)$ time.*

Proof. We know from (7.4) that the algorithm terminates in at most C iterations of the `While` loop. We therefore consider the amount of work involved in one iteration when the current flow is f.

The residual graph G_f has at most $2m$ edges, since each edge of G gives rise to at most two edges in the residual graph. We will maintain G_f using an adjacency list representation; we will have two linked lists for each node v, one containing the edges entering v, and one containing the edges leaving v. To find an s-t path in G_f, we can use breadth-first search or depth-first search,

which run in $O(m+n)$ time; by our assumption that $m \geq n/2$, $O(m+n)$ is the same as $O(m)$. The procedure `augment`(f, P) takes time $O(n)$, as the path P has at most $n-1$ edges. Given the new flow f', we can build the new residual graph in $O(m)$ time: For each edge e of G, we construct the correct forward and backward edges in $G_{f'}$. ■

A somewhat more efficient version of the algorithm would maintain the linked lists of edges in the residual graph G_f as part of the `augment` procedure that changes the flow f via augmentation.

7.2 Maximum Flows and Minimum Cuts in a Network

We now continue with the analysis of the Ford-Fulkerson Algorithm, an activity that will occupy this whole section. In the process, we will not only learn a lot about the algorithm, but also find that analyzing the algorithm provides us with considerable insight into the Maximum-Flow Problem itself.

Analyzing the Algorithm: Flows and Cuts

Our next goal is to show that the flow that is returned by the Ford-Fulkerson Algorithm has the maximum possible value of any flow in G. To make progress toward this goal, we return to an issue that we raised in Section 7.1: the way in which the structure of the flow network places upper bounds on the maximum value of an s-t flow. We have already seen one upper bound: the value $v(f)$ of any s-t-flow f is at most $C = \sum_{e \text{ out of } s} c_e$. Sometimes this bound is useful, but sometimes it is very weak. We now use the notion of a *cut* to develop a much more general means of placing upper bounds on the maximum-flow value.

Consider dividing the nodes of the graph into two sets, A and B, so that $s \in A$ and $t \in B$. As in our discussion in Section 7.1, any such division places an upper bound on the maximum possible flow value, since all the flow must cross from A to B somewhere. Formally, we say that an *s-t cut* is a partition (A, B) of the vertex set V, so that $s \in A$ and $t \in B$. The *capacity* of a cut (A, B), which we will denote $c(A, B)$, is simply the sum of the capacities of all edges out of A: $c(A, B) = \sum_{e \text{ out of } A} c_e$.

Cuts turn out to provide very natural upper bounds on the values of flows, as expressed by our intuition above. We make this precise via a sequence of facts.

(7.6) *Let f be any s-t flow, and (A, B) any s-t cut. Then $v(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.*

This statement is actually much stronger than a simple upper bound. It says that by watching the amount of flow f sends across a cut, we can exactly *measure* the flow value: It is the total amount that leaves A, minus the amount that "swirls back" into A. This makes sense intuitively, although the proof requires a little manipulation of sums.

Proof. By definition $v(f) = f^{\text{out}}(s)$. By assumption we have $f^{\text{in}}(s) = 0$, as the source s has no entering edges, so we can write $v(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$. Since every node v in A other than s is internal, we know that $f^{\text{out}}(v) - f^{\text{in}}(v) = 0$ for all such nodes. Thus

$$v(f) = \sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v)),$$

since the only term in this sum that is nonzero is the one in which v is set to s.

Let's try to rewrite the sum on the right as follows. If an edge e has both ends in A, then $f(e)$ appears once in the sum with a "+" and once with a "−", and hence these two terms cancel out. If e has only its tail in A, then $f(e)$ appears just once in the sum, with a "+". If e has only its head in A, then $f(e)$ also appears just once in the sum, with a "−". Finally, if e has neither end in A, then $f(e)$ doesn't appear in the sum at all. In view of this, we have

$$\sum_{v \in A} f^{\text{out}}(v) - f^{\text{in}}(v) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A).$$

Putting together these two equations, we have the statement of (7.6). ■

If $A = \{s\}$, then $f^{\text{out}}(A) = f^{\text{out}}(s)$, and $f^{\text{in}}(A) = 0$ as there are no edges entering the source by assumption. So the statement for this set $A = \{s\}$ is exactly the definition of the flow value $v(f)$.

Note that if (A, B) is a cut, then the edges into B are precisely the edges out of A. Similarly, the edges out of B are precisely the edges into A. Thus we have $f^{\text{out}}(A) = f^{\text{in}}(B)$ and $f^{\text{in}}(A) = f^{\text{out}}(B)$, just by comparing the definitions for these two expressions. So we can rephrase (7.6) in the following way.

(7.7) *Let f be any s-t flow, and (A, B) any s-t cut. Then $v(f) = f^{\text{in}}(B) - f^{\text{out}}(B)$.*

If we set $A = V - \{t\}$ and $B = \{t\}$ in (7.7), we have $v(f) = f^{\text{in}}(B) - f^{\text{out}}(B) = f^{\text{in}}(t) - f^{\text{out}}(t)$. By our assumption the sink t has no leaving edges, so we have $f^{\text{out}}(t) = 0$. This says that we could have originally defined the *value* of a flow equally well in terms of the sink t: It is $f^{\text{in}}(t)$, the amount of flow arriving at the sink.

A very useful consequence of (7.6) is the following upper bound.

(7.8) *Let f be any s-t flow, and (A, B) any s-t cut. Then $v(f) \leq c(A, B)$.*

Proof.

$$\begin{aligned} v(f) &= f^{\text{out}}(A) - f^{\text{in}}(A) \\ &\leq f^{\text{out}}(A) \\ &= \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c_e \\ &= c(A, B). \end{aligned}$$

Here the first line is simply (7.6); we pass from the first to the second since $f^{\text{in}}(A) \geq 0$, and we pass from the third to the fourth by applying the capacity conditions to each term of the sum. ■

In a sense, (7.8) looks weaker than (7.6), since it is only an inequality rather than an equality. However, it will be extremely useful for us, since its right-hand side is independent of any particular flow f. What (7.8) says is that *the value of every flow is upper-bounded by the capacity of every cut*. In other words, if we exhibit any s-t cut in G of some value c^*, we know immediately by (7.8) that there cannot be an s-t flow in G of value greater than c^*. Conversely, if we exhibit any s-t flow in G of some value v^*, we know immediately by (7.8) that there cannot be an s-t cut in G of value less than v^*.

Analyzing the Algorithm: Max-Flow Equals Min-Cut

Let $\overline{f}$ denote the flow that is returned by the Ford-Fulkerson Algorithm. We want to show that $\overline{f}$ has the maximum possible value of any flow in G, and we do this by the method discussed above: We exhibit an s-t cut (A^*, B^*) for which $v(\overline{f}) = c(A^*, B^*)$. This immediately establishes that $\overline{f}$ has the maximum value of any flow, and that (A^*, B^*) has the minimum capacity of any s-t cut.

The Ford-Fulkerson Algorithm terminates when the flow f has no s-t path in the residual graph G_f. This turns out to be the only property needed for proving its maximality.

(7.9) *If f is an s-t-flow such that there is no s-t path in the residual graph G_f, then there is an s-t cut (A^*, B^*) in G for which $v(f) = c(A^*, B^*)$. Consequently, f has the maximum value of any flow in G, and (A^*, B^*) has the minimum capacity of any s-t cut in G.*

Proof. The statement claims the existence of a cut satisfying a certain desirable property; thus we must now identify such a cut. To this end, let A^* denote the set of all nodes v in G for which there is an s-v path in G_f. Let B^* denote the set of all other nodes: $B^* = V - A^*$.

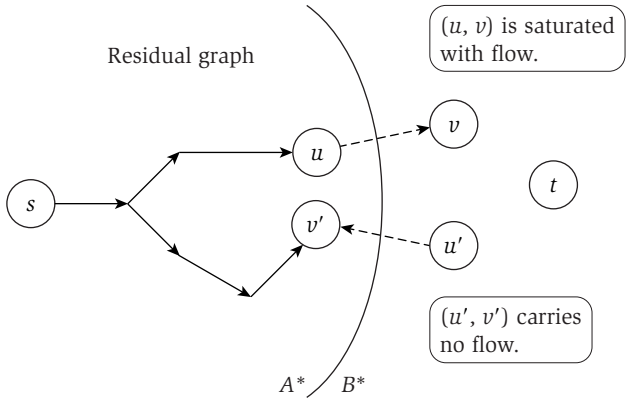

Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

First we establish that (A^*, B^*) is indeed an s-t cut. It is clearly a partition of V. The source s belongs to A^* since there is always a path from s to s. Moreover, $t \notin A^*$ by the assumption that there is no s-t path in the residual graph; hence $t \in B^*$ as desired.

Next, suppose that $e = (u, v)$ is an edge in G for which $u \in A^*$ and $v \in B^*$, as shown in Figure 7.5. We claim that $f(e) = c_e$. For if not, e would be a forward edge in the residual graph G_f, and since $u \in A^*$, there is an s-u path in G_f; appending e to this path, we would obtain an s-v path in G_f, contradicting our assumption that $v \in B^*$.

Now suppose that $e' = (u', v')$ is an edge in G for which $u' \in B^*$ and $v' \in A^*$. We claim that $f(e') = 0$. For if not, e' would give rise to a backward edge $e'' = (v', u')$ in the residual graph G_f, and since $v' \in A^*$, there is an s-v' path in G_f; appending e'' to this path, we would obtain an s-u' path in G_f, contradicting our assumption that $u' \in B^*$.

So all edges out of A^* are completely saturated with flow, while all edges into A^* are completely unused. We can now use (7.6) to reach the desired conclusion:

$$\begin{aligned} v(f) &= f^{\text{out}}(A^*) - f^{\text{in}}(A^*) \\ &= \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) \\ &= \sum_{e \text{ out of } A^*} c_e - 0 \\ &= c(A^*, B^*). \quad \blacksquare \end{aligned}$$

Note how, in retrospect, we can see why the two types of residual edges—forward and backward—are crucial in analyzing the two terms in the expression from (7.6).

Given that the Ford-Fulkerson Algorithm terminates when there is no *s-t* in the residual graph, (7.6) immediately implies its optimality.

(7.10) *The flow $\overline{f}$ returned by the Ford-Fulkerson Algorithm is a maximum flow.*

We also observe that our algorithm can easily be extended to compute a minimum *s-t* cut (A^*, B^*), as follows.

(7.11) *Given a flow f of maximum value, we can compute an s-t cut of minimum capacity in $O(m)$ time.*

Proof. We simply follow the construction in the proof of (7.9). We construct the residual graph G_f, and perform breadth-first search or depth-first search to determine the set A^* of all nodes that s can reach. We then define $B^* = V - A^*$, and return the cut (A^*, B^*). ■

Note that there can be many minimum-capacity cuts in a graph G; the procedure in the proof of (7.11) is simply finding a particular one of these cuts, starting from a maximum flow $\overline{f}$.

As a bonus, we have obtained the following striking fact through the analysis of the algorithm.

(7.12) *In every flow network, there is a flow f and a cut (A, B) so that $v(f) = c(A, B)$.*

The point is that f in (7.12) must be a maximum *s-t* flow; for if there were a flow f' of greater value, the value of f' would exceed the capacity of (A, B), and this would contradict (7.8). Similarly, it follows that (A, B) in (7.12) is a *minimum cut*—no other cut can have smaller capacity—for if there were a cut (A', B') of smaller capacity, it would be less than the value of f, and this again would contradict (7.8). Due to these implications, (7.12) is often called the *Max-Flow Min-Cut Theorem*, and is phrased as follows.

(7.13) *In every flow network, the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut.*

Further Analysis: Integer-Valued Flows

Among the many corollaries emerging from our analysis of the Ford-Fulkerson Algorithm, here is another extremely important one. By (7.2), we maintain an integer-valued flow at all times, and by (7.9), we conclude with a maximum flow. Thus we have

(7.14) *If all capacities in the flow network are integers, then there is a maximum flow f for which every flow value $f(e)$ is an integer.*

Note that (7.14) does not claim that *every* maximum flow is integer-valued, only that *some* maximum flow has this property. Curiously, although (7.14) makes no reference to the Ford-Fulkerson Algorithm, our algorithmic approach here provides what is probably the easiest way to prove it.

Real Numbers as Capacities? Finally, before moving on, we can ask how crucial our assumption of integer capacities was (ignoring (7.4), (7.5) and (7.14), which clearly needed it). First we notice that allowing capacities to be rational numbers does not make the situation any more general, since we can determine the least common multiple of all capacities, and multiply them all by this value to obtain an equivalent problem with integer capacities.

But what if we have real numbers as capacities? Where in the proof did we rely on the capacities being integers? In fact, we relied on it quite crucially: We used (7.2) to establish, in (7.4), that the value of the flow increased by at least 1 in every step. With real numbers as capacities, we should be concerned that the value of our flow keeps increasing, but in increments that become arbitrarily smaller and smaller; and hence we have no guarantee that the number of iterations of the loop is finite. And this turns out to be an extremely real worry, for the following reason: *With pathological choices for the augmenting path, the Ford-Fulkerson Algorithm with real-valued capacities can run forever.*

However, one can still prove that the Max-Flow Min-Cut Theorem (7.12) is true even if the capacities may be real numbers. Note that (7.9) assumed only that the flow f has no s-t path in its residual graph G_f, in order to conclude that there is an s-t cut of equal value. Clearly, for any flow f of maximum value, the residual graph has no s-t-path; otherwise there would be a way to increase the value of the flow. So one can prove (7.12) in the case of real-valued capacities by simply establishing that for every flow network, there exists a maximum flow.

Of course, the capacities in any practical application of network flow would be integers or rational numbers. However, the problem of pathological choices for the augmenting paths can manifest itself even with integer capacities: It can make the Ford-Fulkerson Algorithm take a gigantic number of iterations.

In the next section, we discuss how to select augmenting paths so as to avoid the potential bad behavior of the algorithm.

7.3 Choosing Good Augmenting Paths

In the previous section, we saw that any way of choosing an augmenting path increases the value of the flow, and this led to a bound of C on the number of augmentations, where $C = \sum_{e \text{ out of } s} c_e$. When C is not very large, this can be a reasonable bound; however, it is very weak when C is large.

To get a sense for how bad this bound can be, consider the example graph in Figure 7.2; but this time assume the capacities are as follows: The edges (s, v), (s, u), (v, t) and (u, t) have capacity 100, and the edge (u, v) has capacity 1, as shown in Figure 7.6. It is easy to see that the maximum flow has value 200, and has $f(e) = 100$ for the edges (s, v), (s, u), (v, t) and (u, t) and value 0 on the edge (u, v). This flow can be obtained by a sequence of two augmentations, using the paths of nodes s, u, t and path s, v, t. But consider how bad the Ford-Fulkerson Algorithm can be with pathological choices for the augmenting paths. Suppose we start with augmenting path P_1 of nodes s, u, v, t in this order (as shown in Figure 7.6). This path has $\texttt{bottleneck}(P_1, f) = 1$. After this augmentation, we have $f(e) = 1$ on the edge $e = (u, v)$, so the reverse edge is in the residual graph. For the next augmenting path, we choose the path P_2 of the nodes s, v, u, t in this order. In this second augmentation, we get $\texttt{bottleneck}(P_2, f) = 1$ as well. After this second augmentation, we have $f(e) = 0$ for the edge $e = (u, v)$, so the edge is again in the residual graph. Suppose we alternate between choosing P_1 and P_2 for augmentation. In this case, each augmentation will have 1 as the bottleneck capacity, and it will take 200 augmentations to get the desired flow of value 200. This is exactly the bound we proved in (7.4), since $C = 200$ in this example.

Designing a Faster Flow Algorithm

The goal of this section is to show that with a better choice of paths, we can improve this bound significantly. A large amount of work has been devoted to finding good ways of choosing augmenting paths in the Maximum-Flow Problem so as to minimize the number of iterations. We focus here on one of the most natural approaches and will mention other approaches at the end of the section. Recall that augmentation increases the value of the maximum flow by the bottleneck capacity of the selected path; so if we choose paths with large bottleneck capacity, we will be making a lot of progress. A natural idea is to select the path that has the largest bottleneck capacity. Having to find such paths can slow down each individual iteration by quite a bit. We will avoid this slowdown by not worrying about selecting the path that has *exactly*

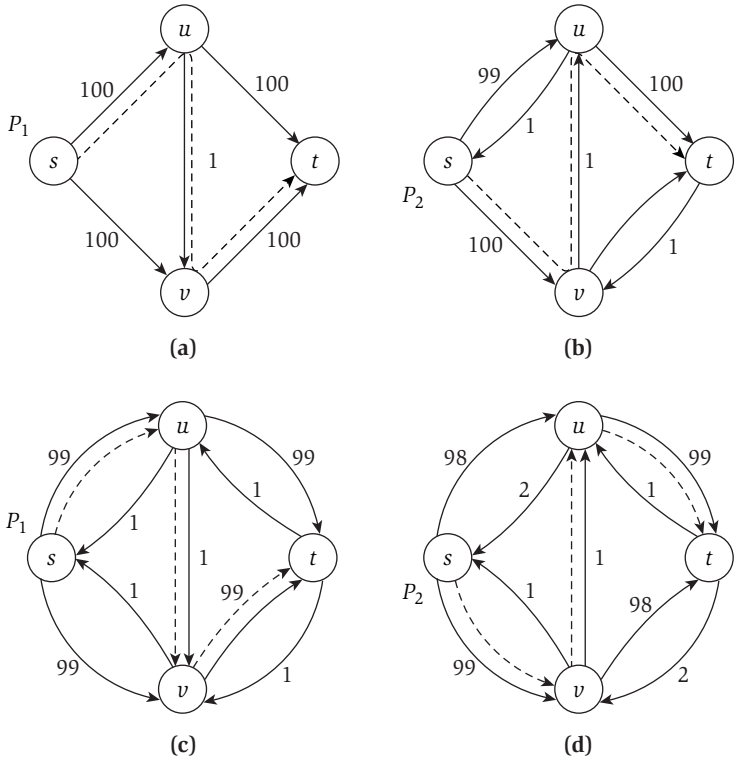

Figure 7.6 Parts (a) through (d) depict four iterations of the Ford-Fulkerson Algorithm using a bad choice of augmenting paths: The augmentations alternate between the path P_1 through the nodes s, u, v, t in order and the path P_2 through the nodes s, v, u, t in order.

the largest bottleneck capacity. Instead, we will maintain a so-called *scaling parameter* Δ, and we will look for paths that have bottleneck capacity of at least Δ.

Let $G_f(\Delta)$ be the subset of the residual graph consisting only of edges with residual capacity of at least Δ. We will work with values of Δ that are powers of 2. The algorithm is as follows.

```
Scaling Max-Flow
  Initially f(e) = 0 for all e in G
  Initially set Δ to be the largest power of 2 that is no larger
        than the maximum capacity out of s: Δ ≤ max_{e out of s} c_e
    While Δ ≥ 1
       While there is an s-t path in the graph G_f(Δ)
          Let P be a simple s-t path in G_f(Δ)
```

```
            f' = augment(f, P)
            Update f to be f' and update G_f(Δ)
        Endwhile
        Δ = Δ/2
    Endwhile
Return f
```

Analyzing the Algorithm

First observe that the new Scaling Max-Flow Algorithm is really just an implementation of the original Ford-Fulkerson Algorithm. The new loops, the value Δ, and the restricted residual graph $G_f(\Delta)$ are only used to guide the selection of residual path—with the goal of using edges with large residual capacity for as long as possible. Hence all the properties that we proved about the original Max-Flow Algorithm are also true for this new version: the flow remains integer-valued throughout the algorithm, and hence all residual capacities are integer-valued.

(7.15) *If the capacities are integer-valued, then throughout the Scaling Max-Flow Algorithm the flow and the residual capacities remain integer-valued. This implies that when $\Delta = 1$, $G_f(\Delta)$ is the same as G_f, and hence when the algorithm terminates the flow, f is of maximum value.*

Next we consider the running time. We call an iteration of the outside `While` loop—with a fixed value of Δ—the Δ-*scaling phase*. It is easy to give an upper bound on the number of different Δ-scaling phases, in terms of the value $C = \sum_{e \text{ out of } s} c_e$ that we also used in the previous section. The initial value of Δ is at most C, it drops by factors of 2, and it never gets below 1. Thus,

(7.16) *The number of iterations of the outer* `While` *loop is at most* $1 + \lceil \log_2 C \rceil$.

The harder part is to bound the number of augmentations done in each scaling phase. The idea here is that we are using paths that augment the flow by a lot, and so there should be relatively few augmentations. During the Δ-scaling phase, we only use edges with residual capacity of at least Δ. Using (7.3), we have

(7.17) *During the Δ-scaling phase, each augmentation increases the flow value by at least Δ.*

The key insight is that at the end of the Δ-scaling phase, the flow f cannot be too far from the maximum possible value.

(7.18) *Let f be the flow at the end of the Δ-scaling phase. There is an s-t cut (A, B) in G for which $c(A, B) \leq v(f) + m\Delta$, where m is the number of edges in the graph G. Consequently, the maximum flow in the network has value at most $v(f) + m\Delta$.*

Proof. This proof is analogous to our proof of (7.9), which established that the flow returned by the original Max-Flow Algorithm is of maximum value.

As in that proof, we must identify a cut (A, B) with the desired property. Let A denote the set of all nodes v in G for which there is an s-v path in $G_f(\Delta)$. Let B denote the set of all other nodes: $B = V - A$. We can see that (A, B) is indeed an s-t cut as otherwise the phase would not have ended.

Now consider an edge $e = (u, v)$ in G for which $u \in A$ and $v \in B$. We claim that $c_e < f(e) + \Delta$. For if this were not the case, then e would be a forward edge in the graph $G_f(\Delta)$, and since $u \in A$, there is an s-u path in $G_f(\Delta)$; appending e to this path, we would obtain an s-v path in $G_f(\Delta)$, contradicting our assumption that $v \in B$. Similarly, we claim that for any edge $e' = (u', v')$ in G for which $u' \in B$ and $v' \in A$, we have $f(e') < \Delta$. Indeed, if $f(e') \geq \Delta$, then e' would give rise to a backward edge $e'' = (v', u')$ in the graph $G_f(\Delta)$, and since $v' \in A$, there is an s-v' path in $G_f(\Delta)$; appending e'' to this path, we would obtain an s-u' path in $G_f(\Delta)$, contradicting our assumption that $u' \in B$.

So all edges e out of A are almost saturated—they satisfy $c_e < f(e) + \Delta$—and all edges into A are almost empty—they satisfy $f(e) < \Delta$. We can now use (7.6) to reach the desired conclusion:

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) \\ &\geq \sum_{e \text{ out of } A} (c_e - \Delta) - \sum_{e \text{ into } A} \Delta \\ &= \sum_{e \text{ out of } A} c_e - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ into } A} \Delta \\ &\geq c(A, B) - m\Delta. \end{aligned}$$

Here the first inequality follows from our bounds on the flow values of edges across the cut, and the second inequality follows from the simple fact that the graph only contains m edges total.

The maximum-flow value is bounded by the capacity of any cut by (7.8). We use the cut (A, B) to obtain the bound claimed in the second statement.

■

(7.19) *The number of augmentations in a scaling phase is at most $2m$.*

Proof. The statement is clearly true in the first scaling phase: we can use each of the edges out of s only for at most one augmentation in that phase. Now consider a later scaling phase Δ, and let f_p be the flow at the end of the *previous* scaling phase. In that phase, we used $\Delta' = 2\Delta$ as our parameter. By (7.18), the maximum flow f^* is at most $v(f^*) \leq v(f_p) + m\Delta' = v(f_p) + 2m\Delta$. In the Δ-scaling phase, each augmentation increases the flow by at least Δ, and hence there can be at most $2m$ augmentations. ■

An augmentation takes $O(m)$ time, including the time required to set up the graph and find the appropriate path. We have at most $1 + \lceil \log_2 C \rceil$ scaling phases and at most $2m$ augmentations in each scaling phase. Thus we have the following result.

(7.20) *The Scaling Max-Flow Algorithm in a graph with m edges and integer capacities finds a maximum flow in at most $2m(1 + \lceil \log_2 C \rceil)$ augmentations. It can be implemented to run in at most $O(m^2 \log_2 C)$ time.*

When C is large, this time bound is much better than the $O(mC)$ bound that applied to an arbitrary implementation of the Ford-Fulkerson Algorithm. In our example at the beginning of this section, we had capacities of size 100, but we could just as well have used capacities of size 2^{100}; in this case, the generic Ford-Fulkerson Algorithm could take time proportional to 2^{100}, while the scaling algorithm will take time proportional to $\log_2(2^{100}) = 100$. One way to view this distinction is as follows: The generic Ford-Fulkerson Algorithm requires time proportional to the *magnitude* of the capacities, while the scaling algorithm only requires time proportional to the number of *bits* needed to specify the capacities in the input to the problem. As a result, the scaling algorithm is running in time polynomial in the size of the input (i.e., the number of edges and the numerical representation of the capacities), and so it meets our traditional goal of achieving a polynomial-time algorithm. Bad implementations of the Ford-Fulkerson Algorithm, which can require close to C iterations, do not meet this standard of polynomiality. (Recall that in Section 6.4 we used the term *pseudo-polynomial* to describe such algorithms, which are polynomial in the magnitudes of the input numbers but not in the number of bits needed to represent them.)

Extensions: Strongly Polynomial Algorithms

Could we ask for something qualitatively better than what the scaling algorithm guarantees? Here is one thing we could hope for: Our example graph (Figure 7.6) had four nodes and five edges; so it would be nice to use a

number of iterations that is polynomial in the numbers 4 and 5, completely independently of the values of the capacities. Such an algorithm, which is polynomial in $|V|$ and $|E|$ only, and works with numbers having a polynomial number of bits, is called a *strongly polynomial algorithm*. In fact, there is a simple and natural implementation of the Ford-Fulkerson Algorithm that leads to such a strongly polynomial bound: each iteration chooses the augmenting path with the fewest number of edges. Dinitz, and independently Edmonds and Karp, proved that with this choice the algorithm terminates in at most $O(mn)$ iterations. In fact, these were the first polynomial algorithms for the Maximum-Flow Problem. There has since been a huge amount of work devoted to improving the running times of maximum-flow algorithms. There are currently algorithms that achieve running times of $O(mn \log n)$, $O(n^3)$, and $O(\min(n^{2/3}, m^{1/2})m \log n \log U)$, where the last bound assumes that all capacities are integral and at most U. In the next section, we'll discuss a strongly polynomial maximum-flow algorithm based on a different principle.

* 7.4 The Preflow-Push Maximum-Flow Algorithm

From the very beginning, our discussion of the Maximum-Flow Problem has been centered around the idea of an augmenting path in the residual graph. However, there are some very powerful techniques for maximum flow that are not explicitly based on augmenting paths. In this section we study one such technique, the Preflow-Push Algorithm.

Designing the Algorithm

Algorithms based on augmenting paths maintain a flow f, and use the `augment` procedure to increase the value of the flow. By way of contrast, the Preflow-Push Algorithm will, in essence, increase the flow on an edge-by-edge basis. Changing the flow on a single edge will typically violate the conservation condition, and so the algorithm will have to maintain something less well behaved than a flow—something that does not obey conservation—as it operates.

Preflows We say that an *s-t preflow* (*preflow*, for short) is a function f that maps each edge e to a nonnegative real number, $f : E \to \mathbf{R}^+$. A preflow f must satisfy the capacity conditions:

(i) For each $e \in E$, we have $0 \leq f(e) \leq c_e$.

In place of the conservation conditions, we require only inequalities: Each node other than s must have at least as much flow entering as leaving.

(ii) For each node v other than the source s, we have

$$\sum_{e \text{ into } v} f(e) \geq \sum_{e \text{ out of } v} f(e).$$

We will call the difference

$$e_f(v) = \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e)$$

the *excess* of the preflow at node v. Notice that a preflow where all nodes other than s and t have zero excess is a flow, and the value of the flow is exactly $e_f(t) = -e_f(s)$. We can still define the concept of a residual graph G_f for a preflow f, just as we did for a flow. The algorithm will "push" flow along edges of the residual graph (using both forward and backward edges).

Preflows and Labelings The Preflow-Push Algorithm will maintain a preflow and work on converting the preflow into a flow. The algorithm is based on the physical intuition that flow naturally finds its way "downhill." The "heights" for this intuition will be labels $h(v)$ for each node v that the algorithm will define and maintain, as shown in Figure 7.7. We will push flow from nodes with higher labels to those with lower labels, following the intuition that fluid flows downhill. To make this precise, a *labeling* is a function $h : V \to \mathbf{Z}_{\geq 0}$ from the nodes to the nonnegative integers. We will also refer to the labels as *heights* of the nodes. We will say that a labeling h and an s-t preflow f are *compatible* if

(i) (*Source and sink conditions*) $h(t) = 0$ and $h(s) = n$,

(ii) (*Steepness conditions*) For all edges $(v, w) \in E_f$ in the residual graph, we have $h(v) \leq h(w) + 1$.

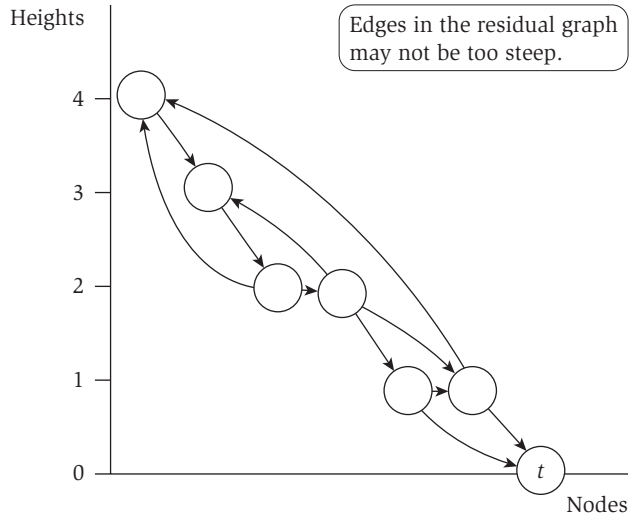

Figure 7.7 A residual graph and a compatible labeling. No edge in the residual graph can be too "steep"—its tail can be at most one unit above its head in height. The source node s must have $h(s) = n$ and is not drawn in the figure.

Intuitively, the height difference n between the source and the sink is meant to ensure that the flow starts high enough to flow from s toward the sink t, while the steepness condition will help by making the descent of the flow gradual enough to make it to the sink.

The key property of a compatible preflow and labeling is that there can be no s-t path in the residual graph.

(7.21) *If s-t preflow f is compatible with a labeling h, then there is no s-t path in the residual graph G_f.*

Proof. We prove the statement by contradiction. Let P be a simple s-t path in the residual graph G. Assume that the nodes along P are $s, v_1, \ldots, v_k = t$. By definition of a labeling compatible with preflow f, we have that $h(s) = n$. The edge (s, v_1) is in the residual graph, and hence $h(v_1) \geq h(s) - 1 = n - 1$. Using induction on i and the steepness condition for the edge (v_{i-1}, v_i), we get that for all nodes v_i in path P the height is at least $h(v_i) \geq n - i$. Notice that the last node of the path is $v_k = t$; hence we get that $h(t) \geq n - k$. However, $h(t) = 0$ by definition; and $k < n$ as the path P is simple. This contradiction proves the claim. ■

Recall from (7.9) that if there is no s-t path in the residual graph G_f of a *flow* f, then the flow has maximum value. This implies the following corollary.

(7.22) *If s-t flow f is compatible with a labeling h, then f is a flow of maximum value.*

Note that (7.21) applies to preflows, while (7.22) is more restrictive in that it applies only to flows. Thus the Preflow-Push Algorithm will maintain a preflow f and a labeling h compatible with f, and it will work on modifying f and h so as to move f toward being a flow. Once f actually becomes a flow, we can invoke (7.22) to conclude that it is a maximum flow. In light of this, we can view the Preflow-Push Algorithm as being in a way orthogonal to the Ford-Fulkerson Algorithm. The Ford-Fulkerson Algorithm maintains a feasible flow while changing it gradually toward optimality. The Preflow-Push Algorithm, on the other hand, maintains a condition that would imply the optimality of a preflow f, *if it were to be a feasible flow*, and the algorithm gradually transforms the preflow f into a flow.

To start the algorithm, we will need to define an initial preflow f and labeling h that are compatible. We will use $h(v) = 0$ for all $v \neq s$, and $h(s) = n$, as our initial labeling. To make a preflow f compatible with this labeling, we need to make sure that no edges leaving s are in the residual graph (as these edges do not satisfy the steepness condition). To this end, we define the initial

preflow as $f(e) = c_e$ for all edges $e = (s, v)$ leaving the source, and $f(e) = 0$ for all other edges.

(7.23) *The initial preflow f and labeling h are compatible.*

Pushing and Relabeling Next we will discuss the steps the algorithm makes toward turning the preflow f into a feasible flow, while keeping it compatible with some labeling h. Consider any node v that has excess—that is, $e_f(v) > 0$. If there is any edge e in the residual graph G_f that leaves v and goes to a node w at a lower height (note that $h(w)$ is at most 1 less than $h(v)$ due to the steepness condition), then we can modify f by pushing some of the excess flow from v to w. We will call this a *push* operation.

```
push(f, h, v, w)
  Applicable if e_f(v) > 0, h(w) < h(v) and (v, w) ∈ E_f
  If e = (v, w) is a forward edge then
    let δ = min(e_f(v), c_e − f(e)) and
    increase f(e) by δ
  If (v, w) is a backward edge then
    let e = (w, v), δ = min(e_f(v), f(e)) and
    decrease f(e) by δ
  Return(f, h)
```

If we cannot push the excess of v along any edge leaving v, then we will need to raise v's height. We will call this a *relabel* operation.

```
relabel(f, h, v)
  Applicable if e_f(v) > 0, and
     for all edges (v, w) ∈ E_f we have h(w) ≥ h(v)
  Increase h(v) by 1
  Return(f, h)
```

The Full Preflow-Push Algorithm So, in summary, the Preflow-Push Algorithm is as follows.

```
Preflow-Push
  Initially h(v) = 0 for all v ≠ s and h(s) = n and
  f(e) = c_e for all e = (s, v) and f(e) = 0 for all other edges
  While there is a node v ≠ t with excess e_f(v) > 0
    Let v be a node with excess
    If there is w such that push(f, h, v, w) can be applied then
       push(f, h, v, w)
```

```
  Else
     relabel(f, h, v)
Endwhile
Return(f)
```

Analyzing the Algorithm

As usual, this algorithm is somewhat underspecified. For an implementation of the algorithm, we will have to specify which node with excess to choose, and how to efficiently select an edge on which to push. However, it is clear that each iteration of this algorithm can be implemented in polynomial time. (We'll discuss later how to implement it reasonably efficiently.) Further, it is not hard to see that the preflow f and the labeling h are compatible throughout the algorithm. If the algorithm terminates—something that is far from obvious based on its description—then there are no nodes other than t with positive excess, and hence the preflow f is in fact a flow. It then follows from (7.22) that f would be a maximum flow at termination.

We summarize a few simple observations about the algorithm.

(7.24) *Throughout the Preflow-Push Algorithm:*

(i) the labels are nonnegative integers;

(ii) f is a preflow, and if the capacities are integral, then the preflow f is integral; and

(iii) the preflow f and labeling h are compatible.

If the algorithm returns a preflow f, then f is a flow of maximum value.

Proof. By (7.23) the initial preflow f and labeling h are compatible. We will show using induction on the number of `push` and `relabel` operations that f and h satisfy the properties of the statement. The `push` operation modifies the preflow f, but the bounds on δ guarantee that the f returned satisfies the capacity constraints, and that excesses all remain nonnegative, so f is a preflow. To see that the preflow f and the labeling h are compatible, note that `push`(f, h, v, w) can add one edge to the residual graph, the reverse edge (v, w), and this edge does satisfy the steepness condition. The `relabel` operation increases the label of v, and hence increases the steepness of all edges leaving v. However, it only applies when no edge leaving v in the residual graph is going downward, and hence the preflow f and the labeling h are compatible after relabeling.

The algorithm terminates if no node other than s or t has excess. In this case, f is a flow by definition; and since the preflow f and the labeling h

remain compatible throughout the algorithm, (7.22) implies that f is a flow of maximum value. ■

Next we will consider the number of `push` and `relabel` operations. First we will prove a limit on the `relabel` operations, and this will help prove a limit on the maximum number of `push` operations possible. The algorithm never changes the label of s (as the source never has positive excess). Each other node v starts with $h(v) = 0$, and its label increases by 1 every time it changes. So we simply need to give a limit on how high a label can get. We only consider a node v for `relabel` when v has excess. The only source of flow in the network is the source s; hence, intuitively, the excess at v must have originated at s. The following consequence of this fact will be key to bounding the labels.

(7.25) *Let f be a preflow. If the node v has excess, then there is a path in G_f from v to the source s.*

Proof. Let A denote all the nodes w such that there is a path from w to s in the residual graph G_f, and let $B = V - A$. We need to show that all nodes with excess are in A.

Notice that $s \in A$. Further, no edges $e = (x, y)$ leaving A can have positive flow, as an edge with $f(e) > 0$ would give rise to a reverse edge (y, x) in the residual graph, and then y would have been in A. Now consider the sum of excesses in the set B, and recall that each node in B has nonnegative excess, as $s \notin B$.

$$0 \leq \sum_{v \in B} e_f(v) = \sum_{v \in B} (f^{\text{in}}(v) - f^{\text{out}}(v))$$

Let's rewrite the sum on the right as follows. If an edge e has both ends in B, then $f(e)$ appears once in the sum with a "$+$" and once with a "$-$", and hence these two terms cancel out. If e has only its head in B, then e leaves A, and we saw above that all edges leaving A have $f(e) = 0$. If e has only its tail in B, then $f(e)$ appears just once in the sum, with a "$-$". So we get

$$0 \leq \sum_{v \in B} e_f(v) = -f^{\text{out}}(B).$$

Since flows are nonnegative, we see that the sum of the excesses in B is zero; since each individual excess in B is nonnegative, they must therefore all be 0. ■

Now we are ready to prove that the labels do not change too much. Recall that n denotes the number of nodes in V.

(7.26) *Throughout the algorithm, all nodes have* $h(v) \leq 2n - 1$.

Proof. The initial labels $h(t) = 0$ and $h(s) = n$ do not change during the algorithm. Consider some other node $v \neq s, t$. The algorithm changes v's label only when applying the `relabel` operation, so let f and h be the preflow and labeling returned by a `relabel`(f, h, v) operation. By (7.25) there is a path P in the residual graph G_f from v to s. Let $|P|$ denote the number of edges in P, and note that $|P| \leq n - 1$. The steepness condition implies that heights of the nodes can decrease by at most 1 along each edge in P, and hence $h(v) - h(s) \leq |P|$, which proves the statement. ■

Labels are monotone increasing throughout the algorithm, so this statement immediately implies a limit on the number of relabeling operations.

(7.27) *Throughout the algorithm, each node is relabeled at most* $2n - 1$ *times, and the total number of relabeling operations is less than* $2n^2$.

Next we will bound the number of `push` operations. We will distinguish two kinds of `push` operations. A `push`(f, h, v, w) operation is *saturating* if either $e = (v, w)$ is a forward edge in E_f and $\delta = c_e - f(e)$, or (v, w) is a backward edge with $e = (w, v)$ and $\delta = f(e)$. In other words, the push is saturating if, after the push, the edge (v, w) is no longer in the residual graph. All other `push` operations will be referred to as *nonsaturating*.

(7.28) *Throughout the algorithm, the number of saturating* `push` *operations is at most* $2nm$.

Proof. Consider an edge (v, w) in the residual graph. After a saturating `push`(f, h, v, w) operation, we have $h(v) = h(w) + 1$, and the edge (v, w) is no longer in the residual graph G_f, as shown in Figure 7.8. Before we can push again along this edge, first we have to push from w to v to make the edge (v, w) appear in the residual graph. However, in order to push from w to v, we first need for w's label to increase by at least 2 (so that w is above v). The label of w can increase by 2 at most $n - 1$ times, so a saturating push from v to w can occur at most n times. Each edge $e \in E$ can give rise to two edges in the residual graph, so overall we can have at most $2nm$ saturating pushes. ■

The hardest part of the analysis is proving a bound on the number of nonsaturating pushes, and this also will be the bottleneck for the theoretical bound on the running time.

(7.29) *Throughout the algorithm, the number of nonsaturating* `push` *operations is at most* $2n^2m$.

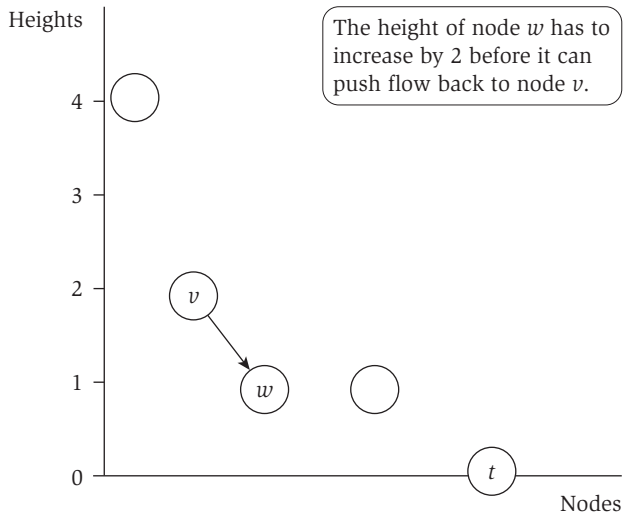

Figure 7.8 After a saturating push(f, h, v, w), the height of v exceeds the height of w by 1.

Proof. For this proof, we will use a so-called *potential function method*. For a preflow f and a compatible labeling h, we define

$$\Phi(f, h) = \sum_{v:e_f(v)>0} h(v)$$

to be the sum of the heights of all nodes with positive excess. (Φ is often called a *potential* since it resembles the "potential energy" of all nodes with positive excess.)

In the initial preflow and labeling, all nodes with positive excess are at height 0, so $\Phi(f, h) = 0$. $\Phi(f, h)$ remains nonnegative throughout the algorithm. A nonsaturating `push`(f, h, v, w) operation decreases $\Phi(f, h)$ by at least 1, since after the push the node v will have no excess, and w, the only node that gets new excess from the operation, is at a height 1 less than v. However, each saturating `push` and each `relabel` operation can increase $\Phi(f, h)$. A `relabel` operation increases $\Phi(f, h)$ by exactly 1. There are at most $2n^2$ `relabel` operations, so the total increase in $\Phi(f, h)$ due to `relabel` operations is $2n^2$. A saturating `push`(f, h, v, w) operation does not change labels, but it can increase $\Phi(f, h)$, since the node w may suddenly acquire positive excess after the push. This would increase $\Phi(f, h)$ by the height of w, which is at most $2n - 1$. There are at most $2nm$ saturating `push` operations, so the total increase in $\Phi(f, h)$ due to `push` operations is at most $2mn(2n - 1)$. So, between the two causes, $\Phi(f, h)$ can increase by at most $4mn^2$ during the algorithm.

But since Φ remains nonnegative throughout, and it decreases by at least 1 on each nonsaturating `push` operation, it follows that there can be at most $4mn^2$ nonsaturating `push` operations. ■

Extensions: An Improved Version of the Algorithm

There has been a lot of work devoted to choosing node selection rules for the Preflow-Push Algorithm to improve the worst-case running time. Here we consider a simple rule that leads to an improved $O(n^3)$ bound on the number of nonsaturating `push` operations.

(7.30) *If at each step we choose the node with excess at maximum height, then the number of nonsaturating* `push` *operations throughout the algorithm is at most* $4n^3$.

Proof. Consider the maximum height $H = \max_{v:e_f(v)>0} h(v)$ of any node with excess as the algorithm proceeds. The analysis will use this maximum height H in place of the potential function Φ in the previous $O(n^2m)$ bound.

This maximum height H can only increase due to relabeling (as flow is always pushed to nodes at lower height), and so the total increase in H throughout the algorithm is at most $2n^2$ by (7.26). H starts out 0 and remains nonnegative, so the number of times H changes is at most $4n^2$.

Now consider the behavior of the algorithm over a phase of time in which H remains constant. We claim that each node can have at most one nonsaturating `push` operation during this phase. Indeed, during this phase, flow is being pushed from nodes at height H to nodes at height $H-1$; and after a nonsaturating `push` operation from v, it must receive flow from a node at height $H+1$ before we can push from it again.

Since there are at most n nonsaturating `push` operations between each change to H, and H changes at most $4n^2$ times, the total number of nonsaturating `push` operations is at most $4n^3$. ■

As a follow-up to (7.30), it is interesting to note that experimentally the computational bottleneck of the method is the number of relabeling operations, and a better experimental running time is obtained by variants that work on increasing labels faster than one by one. This is a point that we pursue further in some of the exercises.

Implementing the Preflow-Push Algorithm

Finally, we need to briefly discuss how to implement this algorithm efficiently. Maintaining a few simple data structures will allow us to effectively implement

the operations of the algorithm in constant time each, and overall to implement the algorithm in time $O(mn)$ plus the number of nonsaturating `push` operations. Hence the generic algorithm will run in $O(mn^2)$ time, while the version that always selects the node at maximum height will run in $O(n^3)$ time.

We can maintain all nodes with excess on a simple list, and so we will be able to select a node with excess in constant time. One has to be a bit more careful to be able to select a node with maximum height H in constant time. In order to do this, we will maintain a linked list of all nodes with excess at every possible height. Note that whenever a node v gets relabeled, or continues to have positive excess after a `push`, it remains a node with maximum height H. Thus we only have to select a new node after a `push` when the current node v no longer has positive excess. If node v was at height H, then the new node at maximum height will also be at height H or, if no node at height H has excess, then the maximum height will be $H - 1$, since the previous `push` operation out of v pushed flow to a node at height $H - 1$.

Now assume we have selected a node v, and we need to select an edge (v, w) on which to apply `push`(f, h, v, w) (or `relabel`(f, h, v) if no such w exists). To be able to select an edge quickly, we will use the adjacency list representation of the graph. More precisely, we will maintain, for each node v, all possible edges leaving v in the residual graph (both forward and backward edges) in a linked list, and with each edge we keep its capacity and flow value. Note that this way we have two copies of each edge in our data structure: a forward and a backward copy. These two copies will have pointers to each other, so that updates done at one copy can be carried over to the other one in $O(1)$ time. We will select edges leaving a node v for `push` operations in the order they appear on node v's list. To facilitate this selection, we will maintain a pointer `current`(v) for each node v to the last edge on the list that has been considered for a `push` operation. So, if node v no longer has excess after a nonsaturating `push` operation out of node v, the pointer `current`(v) will stay at this edge, and we will use the same edge for the next `push` operation out of v. After a saturating `push` operation out of node v, we advance `current`(v) to the next edge on the list.

The key observation is that, after advancing the pointer `current`(v) from an edge (v, w), we will not want to apply `push` to this edge again until we relabel v.

(7.31) *After the* `current`(v) *pointer is advanced from an edge* (v, w)*, we cannot apply* `push` *to this edge until* v *gets relabeled.*

Proof. At the moment `current`(v) is advanced from the edge (v, w), there is some reason `push` cannot be applied to this edge. Either $h(w) \geq h(v)$, or the

edge is not in the residual graph. In the first case, we clearly need to relabel v before applying a `push` on this edge. In the latter case, one needs to apply `push` to the reverse edge (w, v) to make (v, w) reenter the residual graph. However, when we apply `push` to edge (w, v), then w is above v, and so v needs to be relabeled before one can push flow from v to w again. ■

Since edges do not have to be considered again for `push` before relabeling, we get the following.

(7.32) *When the* `current`(v) *pointer reaches the end of the edge list for* v*, the* `relabel` *operation can be applied to node* v*.*

After relabeling node v, we reset `current`(v) to the first edge on the list and start considering edges again in the order they appear on v's list.

(7.33) *The running time of the Preflow-Push Algorithm, implemented using the above data structures, is* $O(mn)$ *plus* $O(1)$ *for each nonsaturating* `push` *operation. In particular, the generic Preflow-Push Algorithm runs in* $O(n^2m)$ *time, while the version where we always select the node at maximum height runs in* $O(n^3)$ *time.*

Proof. The initial flow and relabeling is set up in $O(m)$ time. Both `push` and `relabel` operations can be implemented in $O(1)$ time, once the operation has been selected. Consider a node v. We know that v can be relabeled at most $2n$ times throughout the algorithm. We will consider the total time the algorithm spends on finding the right edge on which to `push` flow out of node v, between two times that node v gets relabeled. If node v has d_v adjacent edges, then by (7.32) we spend $O(d_v)$ time on advancing the `current`(v) pointer between consecutive relabelings of v. Thus the total time spent on advancing the `current` pointers throughout the algorithm is $O(\sum_{v \in V} nd_v) = O(mn)$, as claimed. ■

7.5 A First Application: The Bipartite Matching Problem

Having developed a set of powerful algorithms for the Maximum-Flow Problem, we now turn to the task of developing applications of maximum flows and minimum cuts in graphs. We begin with two very basic applications. First, in this section, we discuss the Bipartite Matching Problem mentioned at the beginning of this chapter. In the next section, we discuss the more general *Disjoint Paths Problem*.

The Problem

One of our original goals in developing the Maximum-Flow Problem was to be able to solve the Bipartite Matching Problem, and we now show how to do this. Recall that a *bipartite graph* $G = (V, E)$ is an undirected graph whose node set can be partitioned as $V = X \cup Y$, with the property that every edge $e \in E$ has one end in X and the other end in Y. A *matching* M in G is a subset of the edges $M \subseteq E$ such that each node appears in at most one edge in M. The Bipartite Matching Problem is that of finding a matching in G of largest possible size.

Designing the Algorithm

The graph defining a matching problem is undirected, while flow networks are directed; but it is actually not difficult to use an algorithm for the Maximum-Flow Problem to find a maximum matching.

Beginning with the graph G in an instance of the Bipartite Matching Problem, we construct a flow network G' as shown in Figure 7.9. First we direct all edges in G from X to Y. We then add a node s, and an edge (s, x) from s to each node in X. We add a node t, and an edge (y, t) from each node in Y to t. Finally, we give each edge in G' a capacity of 1.

We now compute a maximum s-t flow in this network G'. We will discover that the value of this maximum is equal to the size of the maximum matching in G. Moreover, our analysis will show how one can use the flow itself to recover the matching.

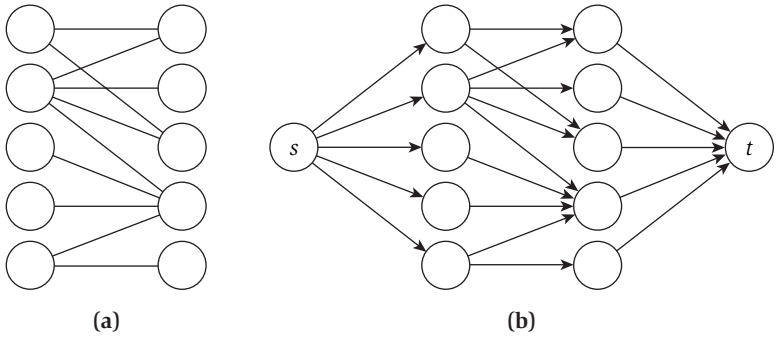

Figure 7.9 (a) A bipartite graph. (b) The corresponding flow network, with all capacities equal to 1.

Analyzing the Algorithm

The analysis is based on showing that integer-valued flows in G' encode matchings in G in a fairly transparent fashion. First, suppose there is a matching in G consisting of k edges $(x_{i_1}, y_{i_1}), \ldots, (x_{i_k}, y_{i_k})$. Then consider the flow f that sends one unit along each path of the form s, x_{i_j}, y_{i_j}, t—that is, $f(e) = 1$ for each edge on one of these paths. One can verify easily that the capacity and conservation conditions are indeed met and that f is an s-t flow of value k.

Conversely, suppose there is a flow f' in G' of value k. By the integrality theorem for maximum flows (7.14), we know there is an integer-valued flow f of value k; and since all capacities are 1, this means that $f(e)$ is equal to either 0 or 1 for each edge e. Now, consider the set M' of edges of the form (x, y) on which the flow value is 1.

Here are three simple facts about the set M'.

(7.34) *M' contains k edges.*

Proof. To prove this, consider the cut (A, B) in G' with $A = \{s\} \cup X$. The value of the flow is the total flow leaving A, minus the total flow entering A. The first of these terms is simply the cardinality of M', since these are the edges leaving A that carry flow, and each carries exactly one unit of flow. The second of these terms is 0, since there are no edges entering A. Thus, M' contains k edges. ■

(7.35) *Each node in X is the tail of at most one edge in M'.*

Proof. To prove this, suppose $x \in X$ were the tail of at least two edges in M'. Since our flow is integer-valued, this means that at least two units of flow leave from x. By conservation of flow, at least two units of flow would have to come into x—but this is not possible, since only a single edge of capacity 1 enters x. Thus x is the tail of at most one edge in M'. ■

By the same reasoning, we can show

(7.36) *Each node in Y is the head of at most one edge in M'.*

Combining these facts, we see that if we view M' as a set of edges in the original bipartite graph G, we get a matching of size k. In summary, we have proved the following fact.

(7.37) *The size of the maximum matching in G is equal to the value of the maximum flow in G'; and the edges in such a matching in G are the edges that carry flow from X to Y in G'.*

Note the crucial way in which the integrality theorem (7.14) figured in this construction: we needed to know if there is a maximum flow in G' that takes only the values 0 and 1.

Bounding the Running Time Now let's consider how quickly we can compute a maximum matching in G. Let $n = |X| = |Y|$, and let m be the number of edges of G. We'll tacitly assume that there is at least one edge incident to each node in the original problem, and hence $m \geq n/2$. The time to compute a maximum matching is dominated by the time to compute an integer-valued maximum flow in G', since converting this to a matching in G is simple. For this flow problem, we have that $C = \sum_{e \text{ out of } s} c_e = |X| = n$, as s has an edge of capacity 1 to each node of X. Thus, by using the $O(mC)$ bound in (7.5), we get the following.

(7.38) *The Ford-Fulkerson Algorithm can be used to find a maximum matching in a bipartite graph in $O(mn)$ time.*

It's interesting that if we were to use the "better" bounds of $O(m^2 \log_2 C)$ or $O(n^3)$ that we developed in the previous sections, we'd get the inferior running times of $O(m^2 \log n)$ or $O(n^3)$ for this problem. There is nothing contradictory in this. These bounds were designed to be good for *all* instances, even when C is very large relative to m and n. But $C = n$ for the Bipartite Matching Problem, and so the cost of this extra sophistication is not needed.

It is worthwhile to consider what the augmenting paths mean in the network G'. Consider the matching M consisting of edges (x_2, y_2), (x_3, y_3), and (x_5, y_5) in the bipartite graph in Figure 7.1; see also Figure 7.10. Let f be the corresponding flow in G'. This matching is not maximum, so f is not a maximum s-t flow, and hence there is an augmenting path in the residual graph G'_f. One such augmenting path is marked in Figure 7.10(b). Note that the edges (x_2, y_2) and (x_3, y_3) are used backward, and all other edges are used forward. All augmenting paths must alternate between edges used backward and forward, as all edges of the graph G' go from X to Y. Augmenting paths are therefore also called *alternating paths* in the context of finding a maximum matching. The effect of this augmentation is to take the edges used backward out of the matching, and replace them with the edges going forward. Because the augmenting path goes from s to t, there is one more forward edge than backward edge; thus the size of the matching increases by one.

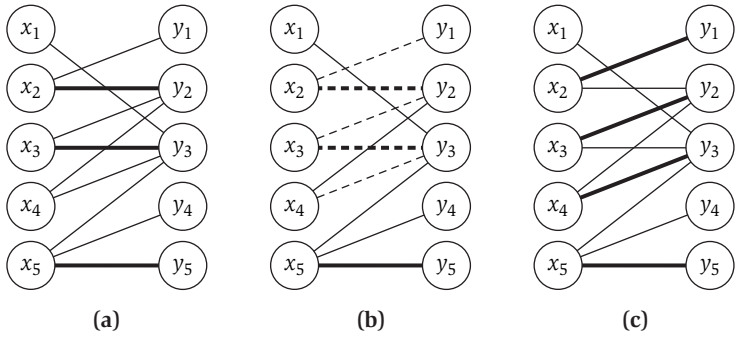

Figure 7.10 (a) A bipartite graph, with a matching M. (b) The augmenting path in the corresponding residual graph. (c) The matching obtained by the augmentation.

Extensions: The Structure of Bipartite Graphs with No Perfect Matching

Algorithmically, we've seen how to find perfect matchings: We use the algorithm above to find a maximum matching and then check to see if this matching is perfect.

But let's ask a slightly less algorithmic question. Not all bipartite graphs have perfect matchings. What does a bipartite graph without a perfect matching look like? Is there an easy way to see that a bipartite graph does not have a perfect matching—or at least an easy way to convince someone the graph has no perfect matching, after we run the algorithm? More concretely, it would be nice if the algorithm, upon concluding that there is no perfect matching, could produce a short "certificate" of this fact. The certificate could allow someone to be quickly convinced that there is no perfect matching, without having to look over a trace of the entire execution of the algorithm.

One way to understand the idea of such a certificate is as follows. We can decide if the graph G has a perfect matching by checking if the maximum flow in a related graph G' has value at least n. By the Max-Flow Min-Cut Theorem, there will be an s-t cut of capacity less than n if the maximum-flow value in G' has value less than n. So, in a way, a cut with capacity less than n provides such a certificate. However, we want a certificate that has a natural meaning in terms of the original graph G.

What might such a certificate look like? For example, if there are nodes $x_1, x_2 \in X$ that have only one incident edge each, and the other end of each edge is the same node y, then clearly the graph has no perfect matching: both x_1 and x_2 would need to get matched to the same node y. More generally, consider a subset of nodes $A \subseteq X$, and let $\Gamma(A) \subseteq Y$ denote the set of all nodes

that are adjacent to nodes in A. If the graph has a perfect matching, then each node in A has to be matched to a different node in $\Gamma(A)$, so $\Gamma(A)$ has to be at least as large as A. This gives us the following fact.

(7.39) *If a bipartite graph $G = (V, E)$ with two sides X and Y has a perfect matching, then for all $A \subseteq X$ we must have $|\Gamma(A)| \geq |A|$.*

This statement suggests a type of certificate demonstrating that a graph does not have a perfect matching: a set $A \subseteq X$ such that $|\Gamma(A)| < |A|$. But is the converse of (7.39) also true? Is it the case that whenever there is no perfect matching, there is a set A like this that proves it? The answer turns out to be yes, provided we add the obvious condition that $|X| = |Y|$ (without which there could certainly not be a perfect matching). This statement is known in the literature as *Hall's Theorem*, though versions of it were discovered independently by a number of different people—perhaps first by König—in the early 1900s. The proof of the statement also provides a way to find such a subset A in polynomial time.

(7.40) *Assume that the bipartite graph $G = (V, E)$ has two sides X and Y such that $|X| = |Y|$. Then the graph G either has a perfect matching or there is a subset $A \subseteq X$ such that $|\Gamma(A)| < |A|$. A perfect matching or an appropriate subset A can be found in $O(mn)$ time.*

Proof. We will use the same graph G' as in (7.37). Assume that $|X| = |Y| = n$. By (7.37) the graph G has a maximum matching if and only if the value of the maximum flow in G' is n.

We need to show that if the value of the maximum flow is less than n, then there is a subset A such that $|\Gamma(A)| < |A|$, as claimed in the statement. By the Max-Flow Min-Cut Theorem (7.12), if the maximum-flow value is less than n, then there is a cut (A', B') with capacity less than n in G'. Now the set A' contains s, and may contain nodes from both X and Y as shown in Figure 7.11. We claim that the set $A = X \cap A'$ has the claimed property. This will prove both parts of the statement, as we've seen in (7.11) that a minimum cut (A', B') can also be found by running the Ford-Fulkerson Algorithm.

First we claim that one can modify the minimum cut (A', B') so as to ensure that $\Gamma(A) \subseteq A'$, where $A = X \cap A'$ as before. To do this, consider a node $y \in \Gamma(A)$ that belongs to B' as shown in Figure 7.11 (a). We claim that by moving y from B' to A', we do not increase the capacity of the cut. For what happens when we move y from B' to A'? The edge (y, t) now crosses the cut, increasing the capacity by one. But previously there was *at least* one edge (x, y) with $x \in A$, since $y \in \Gamma(A)$; all edges from A and y used to cross the cut, and don't anymore. Thus, overall, the capacity of the cut cannot increase. (Note that we

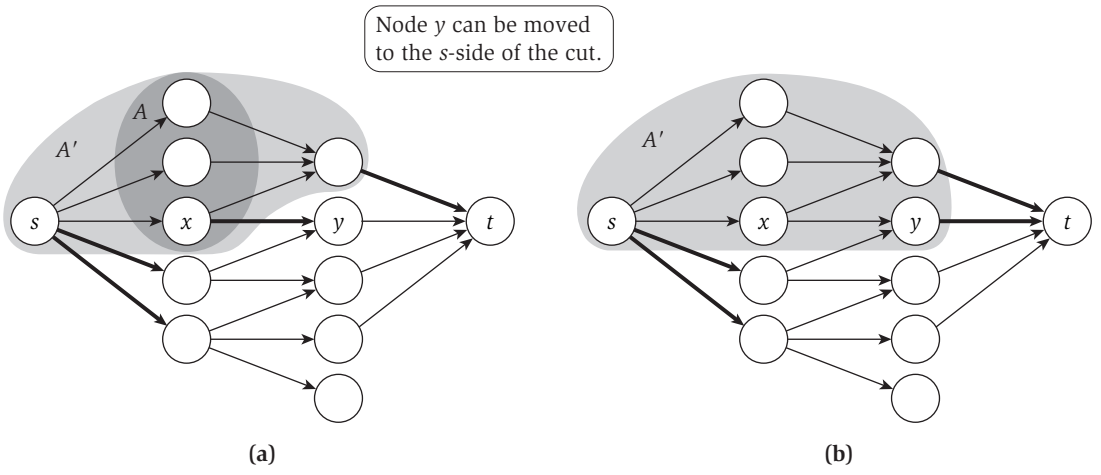

Figure 7.11 (a) A minimum cut in proof of (7.40). (b) The same cut after moving node y to the A' side. The edges crossing the cut are dark.

don't have to be concerned about nodes $x \in X$ that are not in A. The two ends of the edge (x, y) will be on different sides of the cut, but this edge does not add to the capacity of the cut, as it goes from B' to A'.)

Next consider the capacity of this minimum cut (A', B') that has $\Gamma(A) \subseteq A'$ as shown in Figure 7.11(b). Since all neighbors of A belong to A', we see that the only edges out of A' are either edges that leave the source s or that enter the sink t. Thus the capacity of the cut is exactly

$$c(A', B') = |X \cap B'| + |Y \cap A'|.$$

Notice that $|X \cap B'| = n - |A|$, and $|Y \cap A'| \geq |\Gamma(A)|$. Now the assumption that $c(A', B') < n$ implies that

$$n - |A| + |\Gamma(A)| \leq |X \cap B'| + |Y \cap A'| = c(A', B') < n.$$

Comparing the first and the last terms, we get the claimed inequality $|A| > |\Gamma(A)|$. ■

7.6 Disjoint Paths in Directed and Undirected Graphs

In Section 7.1, we described a flow f as a kind of "traffic" in the network. But our actual definition of a flow has a much more static feel to it: For each edge e, we simply specify a number $f(e)$ saying the amount of flow crossing e. Let's see if we can revive the more dynamic, traffic-oriented picture a bit, and try formalizing the sense in which units of flow "travel" from the source to

the sink. From this more dynamic view of flows, we will arrive at something called the *s-t Disjoint Paths Problem*.

The Problem

In defining this problem precisely, we will deal with two issues. First, we will make precise this intuitive correspondence between units of flow traveling along paths, and the notion of flow we've studied so far. Second, we will extend the Disjoint Paths Problem to *undirected* graphs. We'll see that, despite the fact that the Maximum-Flow Problem was defined for a directed graph, it can naturally be used also to handle related problems on undirected graphs.

We say that a set of paths is *edge-disjoint* if their edge sets are disjoint, that is, no two paths share an edge, though multiple paths may go through some of the same nodes. Given a directed graph $G = (V, E)$ with two distinguished nodes $s, t \in V$, the *Directed Edge-Disjoint Paths Problem* is to find the maximum number of edge-disjoint s-t paths in G. The *Undirected Edge-Disjoint Paths Problem* is to find the maximum number of edge-disjoint s-t paths in an undirected graph G. The related question of finding paths that are not only edge-disjoint, but also node-disjoint (of course, other than at nodes s and t) will be considered in the exercises to this chapter.

Designing the Algorithm

Both the directed and the undirected versions of the problem can be solved very naturally using flows. Let's start with the directed problem. Given the graph $G = (V, E)$, with its two distinguished nodes s and t, we define a flow network in which s and t are the source and sink, respectively, and with a capacity of 1 on each edge. Now suppose there are k edge-disjoint s-t paths. We can make each of these paths carry one unit of flow: We set the flow to be $f(e) = 1$ for each edge e on any of the paths, and $f(e') = 0$ on all other edges, and this defines a feasible flow of value k.

(7.41) *If there are k edge-disjoint paths in a directed graph G from s to t, then the value of the maximum s-t flow in G is at least k.*

Suppose we could show the converse to (7.41) as well: If there is a flow of value k, then there exist k edge-disjoint s-t paths. Then we could simply compute a maximum s-t flow in G and declare (correctly) this to be the maximum number of edge-disjoint s-t paths.

We now proceed to prove this converse statement, confirming that this approach using flow indeed gives us the correct answer. Our analysis will also provide a way to extract k edge-disjoint paths from an integer-valued flow sending k units from s to t. Thus computing a maximum flow in G will

not only give us the maximum *number* of edge-disjoint paths, but the paths as well.

Analyzing the Algorithm

Proving the converse direction of (7.41) is the heart of the analysis, since it will immediately establish the optimality of the flow-based algorithm to find disjoint paths.

To prove this, we will consider a flow of value at least k, and construct k edge-disjoint paths. By (7.14), we know that there is a maximum flow f with integer flow values. Since all edges have a capacity bound of 1, and the flow is integer-valued, each edge that carries flow under f has exactly one unit of flow on it. Thus we just need to show the following.

(7.42) *If f is a 0-1 valued flow of value v, then the set of edges with flow value $f(e) = 1$ contains a set of v edge-disjoint paths.*

Proof. We prove this by induction on the number of edges in f that carry flow. If $v = 0$, there is nothing to prove. Otherwise, there must be an edge (s, u) that carries one unit of flow. We now "trace out" a path of edges that must also carry flow: Since (s, u) carries a unit of flow, it follows by conservation that there is some edge (u, v) that carries one unit of flow, and then there must be an edge (v, w) that carries one unit of flow, and so forth. If we continue in this way, one of two things will eventually happen: Either we will reach t, or we will reach a node v for the second time.

If the first case happens—we find a path P from s to t—then we'll use this path as one of our v paths. Let f' be the flow obtained by decreasing the flow values on the edges along P to 0. This new flow f' has value $v - 1$, and it has fewer edges that carry flow. Applying the induction hypothesis for f', we get $v - 1$ edge-disjoint paths, which, along with path P, form the v paths claimed.

If P reaches a node v for the second time, then we have a situation like the one pictured in Figure 7.12. (The edges in the figure all carry one unit of flow, and the dashed edges indicate the path traversed so far, which has just reached a node v for the second time.) In this case, we can make progress in a different way.

Consider the cycle C of edges visited between the first and second appearances of v. We obtain a new flow f' from f by decreasing the flow values on the edges along C to 0. This new flow f' has value v, but it has fewer edges that carry flow. Applying the induction hypothesis for f', we get the v edge-disjoint paths as claimed. ■

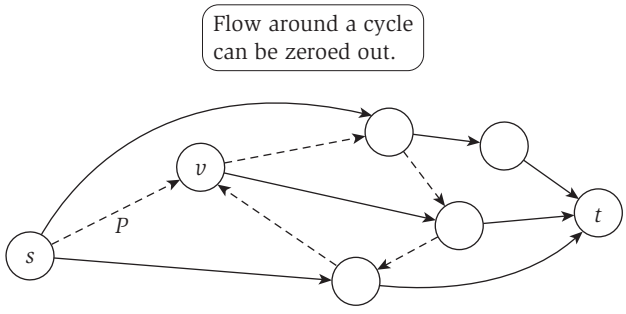

Figure 7.12 The edges in the figure all carry one unit of flow. The path P of dashed edges is one possible path in the proof of (7.42).

We can summarize (7.41) and (7.42) in the following result.

(7.43) *There are k edge-disjoint paths in a directed graph G from s to t if and only if the value of the maximum value of an s-t flow in G is at least k.*

Notice also how the proof of (7.42) provides an actual procedure for constructing the k paths, given an integer-valued maximum flow in G. This procedure is sometimes referred to as a *path decomposition* of the flow, since it "decomposes" the flow into a constituent set of paths. Hence we have shown that our flow-based algorithm finds the maximum number of edge-disjoint s-t paths and also gives us a way to construct the actual paths.

Bounding the Running Time For this flow problem, $C = \sum_{e \text{ out of } s} c_e \leq |V| = n$, as there are at most $|V|$ edges out of s, each of which has capacity 1. Thus, by using the $O(mC)$ bound in (7.5), we get an integer maximum flow in $O(mn)$ time.

The path decomposition procedure in the proof of (7.42), which produces the paths themselves, can also be made to run in $O(mn)$ time. To see this, note that this procedure, with a little care, can produce a single path from s to t using at most constant work per edge in the graph, and hence in $O(m)$ time. Since there can be at most $n - 1$ edge-disjoint paths from s to t (each must use a different edge out of s), it therefore takes time $O(mn)$ to produce all the paths.

In summary, we have shown

(7.44) *The Ford-Fulkerson Algorithm can be used to find a maximum set of edge-disjoint s-t paths in a directed graph G in $O(mn)$ time.*

A Version of the Max-Flow Min-Cut Theorem for Disjoint Paths The Max-Flow Min-Cut Theorem (7.13) can be used to give the following characteri-

zation of the maximum number of edge-disjoint s-t paths. We say that a set $F \subseteq E$ of edges *separates* s from t if, after removing the edges F from the graph G, no s-t paths remain in the graph.

(7.45) *In every directed graph with nodes s and t, the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal separates s from t.*

Proof. If the removal of a set $F \subseteq E$ of edges separates s from t, then each s-t path must use at least one edge from F, and hence the number of edge-disjoint s-t paths is at most $|F|$.

To prove the other direction, we will use the Max-Flow Min-Cut Theorem (7.13). By (7.43) the maximum number of edge-disjoint paths is the value v of the maximum s-t flow. Now (7.13) states that there is an s-t cut (A, B) with capacity v. Let F be the set of edges that go from A to B. Each edge has capacity 1, so $|F| = v$ and, by the definition of an s-t cut, removing these v edges from G separates s from t. ■

This result, then, can be viewed as the natural special case of the Max-Flow Min-Cut Theorem in which all edge capacities are equal to 1. In fact, this special case was proved by Menger in 1927, much before the full Max-Flow Min-Cut Theorem was formulated and proved; for this reason, (7.45) is often called *Menger's Theorem*. If we think about it, the proof of Hall's Theorem (7.40) for bipartite matchings involves a reduction to a graph with unit-capacity edges, and so it can be proved using Menger's Theorem rather than the general Max-Flow Min-Cut Theorem. In other words, Hall's Theorem is really a special case of Menger's Theorem, which in turn is a special case of the Max-Flow Min-Cut Theorem. And the history follows this progression, since they were discovered in this order, a few decades apart.[2]

Extensions: Disjoint Paths in Undirected Graphs

Finally, we consider the disjoint paths problem in an undirected graph G. Despite the fact that our graph G is now undirected, we can use the maximum-flow algorithm to obtain edge-disjoint paths in G. The idea is quite simple: We replace each undirected edge (u, v) in G by two directed edges (u, v) and

[2] In fact, in an interesting retrospective written in 1981, Menger relates his version of the story of how he first explained his theorem to König, one of the independent discoverers of Hall's Theorem. You might think that König, having thought a lot about these problems, would have immediately grasped why Menger's generalization of his theorem was true, and perhaps even considered it obvious. But, in fact, the opposite happened; König didn't believe it could be right and stayed up all night searching for a counterexample. The next day, exhausted, he sought out Menger and asked him for the proof.

(v, u), and in this way create a directed version G' of G. (We may delete the edges into s and out of t, since they are not useful.) Now we want to use the Ford-Fulkerson Algorithm in the resulting directed graph. However, there is an important issue we need to deal with first. Notice that two paths P_1 and P_2 may be edge-disjoint in the directed graph and yet share an edge in the undirected graph G: This happens if P_1 uses directed edge (u, v) while P_2 uses edge (v, u). However, it is not hard to see that there always exists a maximum flow in any network that uses at most *one* out of each pair of oppositely directed edges.

(7.46) *In any flow network, there is a maximum flow f where for all opposite directed edges $e = (u, v)$ and $e' = (v, u)$, either $f(e) = 0$ or $f(e') = 0$. If the capacities of the flow network are integral, then there also is such an integral maximum flow.*

Proof. We consider any maximum flow f, and we modify it to satisfy the claimed condition. Assume $e = (u, v)$ and $e' = (v, u)$ are opposite directed edges, and $f(e) \neq 0$, $f(e') \neq 0$. Let δ be the smaller of these values, and modify f by decreasing the flow value on both e and e' by δ. The resulting flow f' is feasible, has the same value as f, and its value on one of e and e' is 0. ■

Now we can use the Ford-Fulkerson Algorithm and the path decomposition procedure from (7.42) to obtain edge-disjoint paths in the undirected graph G.

(7.47) *There are k edge-disjoint paths in an undirected graph G from s to t if and only if the maximum value of an s-t flow in the directed version G' of G is at least k. Furthermore, the Ford-Fulkerson Algorithm can be used to find a maximum set of disjoint s-t paths in an undirected graph G in $O(mn)$ time.*

The undirected analogue of (7.45) is also true, as in any s-t cut, at most one of the two oppositely directed edges can cross from the s-side to the t-side of the cut (for if one crosses, then the other must go from the t-side to the s-side).

(7.48) *In every undirected graph with nodes s and t, the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal separates s from t.*

7.7 Extensions to the Maximum-Flow Problem

Much of the power of the Maximum-Flow Problem has essentially nothing to do with the fact that it models traffic in a network. Rather, it lies in the fact that many problems with a nontrivial combinatorial search component can

be solved in polynomial time because they can be reduced to the problem of finding a maximum flow or a minimum cut in a directed graph.

Bipartite Matching is a natural first application in this vein; in the coming sections, we investigate a range of further applications. To begin with, we stay with the picture of flow as an abstract kind of "traffic," and look for more general conditions we might impose on this traffic. These more general conditions will turn out to be useful for some of our further applications.

In particular, we focus on two generalizations of maximum flow. We will see that both can be reduced to the basic Maximum-Flow Problem.

The Problem: Circulations with Demands

One simplifying aspect of our initial formulation of the Maximum-Flow Problem is that we had only a single source s and a single sink t. Now suppose that there can be a set S of sources generating flow, and a set T of sinks that can absorb flow. As before, there is an integer capacity on each edge.

With multiple sources and sinks, it is a bit unclear how to decide which source or sink to favor in a maximization problem. So instead of maximizing the flow value, we will consider a problem where sources have fixed *supply* values and sinks have fixed *demand* values, and our goal is to ship flow from nodes with available supply to those with given demands. Imagine, for example, that the network represents a system of highways or railway lines in which we want to ship products from factories (which have supply) to retail outlets (which have demand). In this type of problem, we will not be seeking to maximize a particular value; rather, we simply want to satisfy all the demand using the available supply.

Thus we are given a flow network $G = (V, E)$ with capacities on the edges. Now, associated with each node $v \in V$ is a *demand* d_v. If $d_v > 0$, this indicates that the node v has a *demand* of d_v for flow; the node is a sink, and it wishes to receive d_v units more flow than it sends out. If $d_v < 0$, this indicates that v has a *supply* of $-d_v$; the node is a source, and it wishes to send out $-d_v$ units more flow than it receives. If $d_v = 0$, then the node v is neither a source nor a sink. We will assume that all capacities and demands are integers.

We use S to denote the set of all nodes with negative demand and T to denote the set of all nodes with positive demand. Although a node v in S wants to send out more flow than it receives, it will be okay for it to have flow that enters on incoming edges; it should just be more than compensated by the flow that leaves v on outgoing edges. The same applies (in the opposite direction) to the set T.

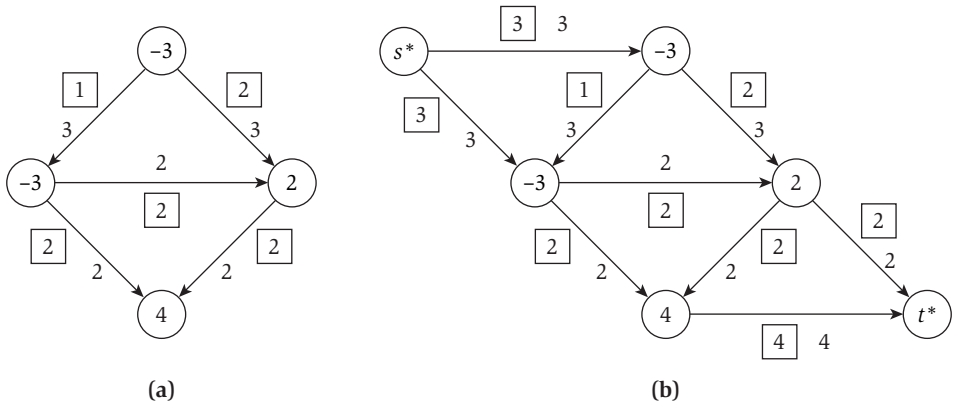

Figure 7.13 (a) An instance of the Circulation Problem together with a solution: Numbers inside the nodes are demands; numbers labeling the edges are capacities and flow values, with the flow values inside boxes. (b) The result of reducing this instance to an equivalent instance of the Maximum-Flow Problem.

In this setting, we say that a *circulation* with demands $\{d_v\}$ is a function f that assigns a nonnegative real number to each edge and satisfies the following two conditions.

(i) *(Capacity conditions)* For each $e \in E$, we have $0 \leq f(e) \leq c_e$.

(ii) *(Demand conditions)* For each $v \in V$, we have v, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

Now, instead of considering a maximization problem, we are concerned with a *feasibility problem*: We want to know whether there *exists* a circulation that meets conditions (i) and (ii).

For example, consider the instance in Figure 7.13(a). Two of the nodes are sources, with demands -3 and -3; and two of the nodes are sinks, with demands 2 and 4. The flow values in the figure constitute a feasible circulation, indicating how all demands can be satisfied while respecting the capacities.

If we consider an arbitrary instance of the Circulation Problem, here is a simple condition that must hold in order for a feasible circulation to exist: The total supply must equal the total demand.

(7.49) *If there exists a feasible circulation with demands $\{d_v\}$, then $\sum_v d_v = 0$.*

Proof. Suppose there exists a feasible circulation f in this setting. Then $\sum_v d_v = \sum_v f^{\text{in}}(v) - f^{\text{out}}(v)$. Now, in this latter expression, the value $f(e)$ for each edge $e = (u, v)$ is counted exactly twice: once in $f^{\text{out}}(u)$ and once in $f^{\text{in}}(v)$. These two terms cancel out; and since this holds for all values $f(e)$, the overall sum is 0. ■

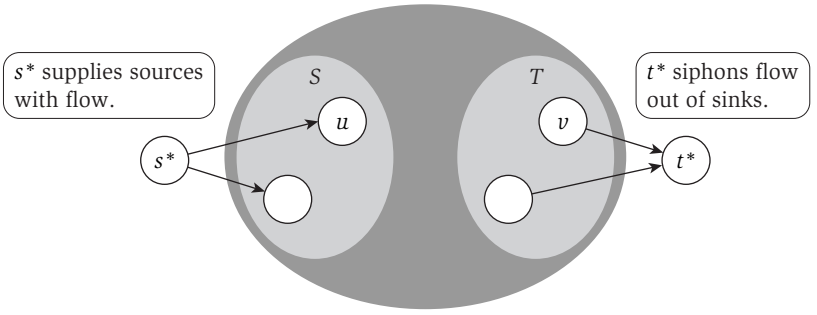

Figure 7.14 Reducing the Circulation Problem to the Maximum-Flow Problem.

Thanks to (7.49), we know that

$$\sum_{v:d_v>0} d_v = \sum_{v:d_v<0} -d_v.$$

Let D denote this common value.

Designing and Analyzing an Algorithm for Circulations

It turns out that we can reduce the problem of finding a feasible circulation with demands $\{d_v\}$ to the problem of finding a maximum s-t flow in a different network, as shown in Figure 7.14.

The reduction looks very much like the one we used for Bipartite Matching: we attach a "super-source" s^* to each node in S, and a "super-sink" t^* to each node in T. More specifically, we create a graph G' from G by adding new nodes s^* and t^* to G. For each node $v \in T$—that is, each node v with $d_v > 0$—we add an edge (v, t^*) with capacity d_v. For each node $u \in S$—that is, each node with $d_u < 0$—we add an edge (s^*, u) with capacity $-d_u$. We carry the remaining structure of G over to G' unchanged.

In this graph G', we will be seeking a maximum s^*-t^* flow. Intuitively, we can think of this reduction as introducing a node s^* that "supplies" all the sources with their extra flow, and a node t^* that "siphons" the extra flow out of the sinks. For example, part (b) of Figure 7.13 shows the result of applying this reduction to the instance in part (a).

Note that there cannot be an s^*-t^* flow in G' of value greater than D, since the cut (A, B) with $A = \{s^*\}$ only has capacity D. Now, if there is a feasible circulation f with demands $\{d_v\}$ in G, then by sending a flow value of $-d_v$ on each edge (s^*, v), and a flow value of d_v on each edge (v, t^*), we obtain an s^*-t^* flow in G' of value D, and so this is a maximum flow. Conversely, suppose there is a (maximum) s^*-t^* flow in G' of value D. It must be that every edge

out of s^*, and every edge into t^*, is completely saturated with flow. Thus, if we delete these edges, we obtain a circulation f in G with $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ for each node v. Further, if there is a flow of value D in G', then there is such a flow that takes integer values.

In summary, we have proved the following.

(7.50) *There is a feasible circulation with demands $\{d_v\}$ in G if and only if the maximum s^*-t^* flow in G' has value D. If all capacities and demands in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer-valued.*

At the end of Section 7.5, we used the Max-Flow Min-Cut Theorem to derive the characterization (7.40) of bipartite graphs that do not have perfect matchings. We can give an analogous characterization for graphs that do not have a feasible circulation. The characterization uses the notion of a *cut*, adapted to the present setting. In the context of circulation problems with demands, a cut (A, B) is any partition of the node set V into two sets, with no restriction on which side of the partition the sources and sinks fall. We include the characterization here without a proof.

(7.51) *The graph G has a feasible circulation with demands $\{d_v\}$ if and only if for all cuts (A, B),*

$$\sum_{v \in B} d_v \leq c(A, B).$$

It is important to note that our network has only a single "kind" of flow. Although the flow is supplied from multiple sources, and absorbed at multiple sinks, we cannot place restrictions on which source will supply the flow to which sink; we have to let our algorithm decide this. A harder problem is the *Multicommodity Flow Problem*; here sink t_i must be supplied with flow that originated at source s_i, for each i. We will discuss this issue further in Chapter 11.

The Problem: Circulations with Demands and Lower Bounds

Finally, let us generalize the previous problem a little. In many applications, we not only want to satisfy demands at various nodes; we also want to force the flow to make use of certain edges. This can be enforced by placing *lower bounds* on edges, as well as the usual upper bounds imposed by edge capacities.

Consider a flow network $G = (V, E)$ with a *capacity* c_e and a *lower bound* ℓ_e on each edge e. We will assume $0 \leq \ell_e \leq c_e$ for each e. As before, each node v will also have a demand d_v, which can be either positive or negative. We will assume that all demands, capacities, and lower bounds are integers.

The given quantities have the same meaning as before, and now a lower bound ℓ_e means that the flow value on e must be *at least* ℓ_e. Thus a circulation in our flow network must satisfy the following two conditions.

(i) *(Capacity conditions)* For each $e \in E$, we have $\ell_e \leq f(e) \leq c_e$.

(ii) *(Demand conditions)* For every $v \in V$, we have $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

As before, we wish to decide whether there exists a *feasible circulation*—one that satisfies these conditions.

Designing and Analyzing an Algorithm with Lower Bounds

Our strategy will be to reduce this to the problem of finding a circulation with demands but no lower bounds. (We've seen that this latter problem, in turn, can be reduced to the standard Maximum-Flow Problem.) The idea is as follows. We know that on each edge e, we need to send at least ℓ_e units of flow. So suppose that we define an initial circulation f_0 simply by $f_0(e) = \ell_e$. f_0 satisfies all the capacity conditions (both lower and upper bounds); but it presumably does not satisfy all the demand conditions. In particular,

$$f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{e \text{ into } v} \ell_e - \sum_{e \text{ out of } v} \ell_e.$$

Let us denote this quantity by L_v. If $L_v = d_v$, then we have satisfied the demand condition at v; but if not, then we need to superimpose a circulation f_1 on top of f_0 that will clear the remaining "imbalance" at v. So we need $f_1^{\text{in}}(v) - f_1^{\text{out}}(v) = d_v - L_v$. And how much capacity do we have with which to do this? Having already sent ℓ_e units of flow on each edge e, we have $c_e - \ell_e$ more units to work with.

These considerations directly motivate the following construction. Let the graph G' have the same nodes and edges, with capacities and demands, but no lower bounds. The capacity of edge e will be $c_e - \ell_e$. The demand of node v will be $d_v - L_v$.

For example, consider the instance in Figure 7.15(a). This is the same as the instance we saw in Figure 7.13, except that we have now given one of the edges a lower bound of 2. In part (b) of the figure, we eliminate this lower bound by sending two units of flow across the edge. This reduces the upper bound on the edge and changes the demands at the two ends of the edge. In the process, it becomes clear that there is no feasible circulation, since after applying the construction there is a node with a demand of -5, and a total of only four units of capacity on its outgoing edges.

We now claim that our general construction produces an equivalent instance with demands but no lower bounds; we can therefore use our algorithm for this latter problem.

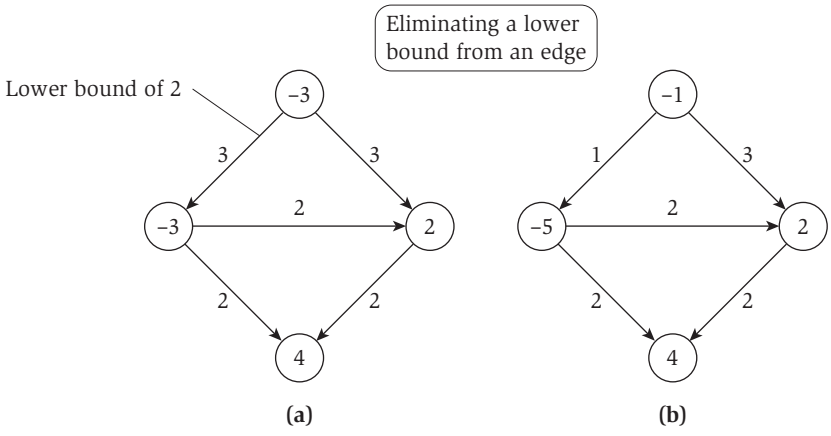

Figure 7.15 (a) An instance of the Circulation Problem with lower bounds: Numbers inside the nodes are demands, and numbers labeling the edges are capacities. We also assign a lower bound of 2 to one of the edges. (b) The result of reducing this instance to an equivalent instance of the Circulation Problem without lower bounds.

(7.52) *There is a feasible circulation in G if and only if there is a feasible circulation in G'. If all demands, capacities, and lower bounds in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer-valued.*

Proof. First suppose there is a circulation f' in G'. Define a circulation f in G by $f(e) = f'(e) + \ell_e$. Then f satisfies the capacity conditions in G, and

$$f^{\text{in}}(v) - f^{\text{out}}(v) = \sum_{e \text{ into } v} (\ell_e + f'(e)) - \sum_{e \text{ out of } v} (\ell_e + f'(e)) = L_v + (d_v - L_v) = d_v,$$

so it satisfies the demand conditions in G as well.

Conversely, suppose there is a circulation f in G, and define a circulation f' in G' by $f'(e) = f(e) - \ell_e$. Then f' satisfies the capacity conditions in G', and

$$(f')^{\text{in}}(v) - (f')^{\text{out}}(v) = \sum_{e \text{ into } v} (f(e) - \ell_e) - \sum_{e \text{ out of } v} (f(e) - \ell_e) = d_v - L_v,$$

so it satisfies the demand conditions in G' as well. ■

7.8 Survey Design

Many problems that arise in applications can, in fact, be solved efficiently by a reduction to Maximum Flow, but it is often difficult to discover when such a reduction is possible. In the next few sections, we give several paradigmatic examples of such problems. The goal is to indicate what such reductions tend

to look like and to illustrate some of the most common uses of flows and cuts in the design of efficient combinatorial algorithms. One point that will emerge is the following: Sometimes the solution one wants involves the computation of a maximum flow, and sometimes it involves the computation of a minimum cut; both flows and cuts are very useful algorithmic tools.

We begin with a basic application that we call *survey design*, a simple version of a task faced by many companies wanting to measure customer satisfaction. More generally, the problem illustrates how the construction used to solve the Bipartite Matching Problem arises naturally in any setting where we want to carefully balance decisions across a set of options—in this case, designing questionnaires by balancing relevant questions across a population of consumers.

The Problem

A major issue in the burgeoning field of *data mining* is the study of consumer preference patterns. Consider a company that sells k products and has a database containing the purchase histories of a large number of customers. (Those of you with "Shopper's Club" cards may be able to guess how this data gets collected.) The company wishes to conduct a survey, sending customized questionnaires to a particular group of n of its customers, to try determining which products people like overall.

Here are the guidelines for designing the survey.

- Each customer will receive questions about a certain subset of the products.
- A customer can only be asked about products that he or she has purchased.
- To make each questionnaire informative, but not too long so as to discourage participation, each customer i should be asked about a number of products between c_i and c_i'.
- Finally, to collect sufficient data about each product, there must be between p_j and p_j' distinct customers asked about each product j.

More formally, the input to the *Survey Design Problem* consists of a bipartite graph G whose nodes are the customers and the products, and there is an edge between customer i and product j if he or she has ever purchased product j. Further, for each customer $i = 1, \ldots, n$, we have limits $c_i \leq c_i'$ on the number of products he or she can be asked about; for each product $j = 1, \ldots, k$, we have limits $p_j \leq p_j'$ on the number of distinct customers that have to be asked about it. The problem is to decide if there is a way to design a questionnaire for each customer so as to satisfy all these conditions.

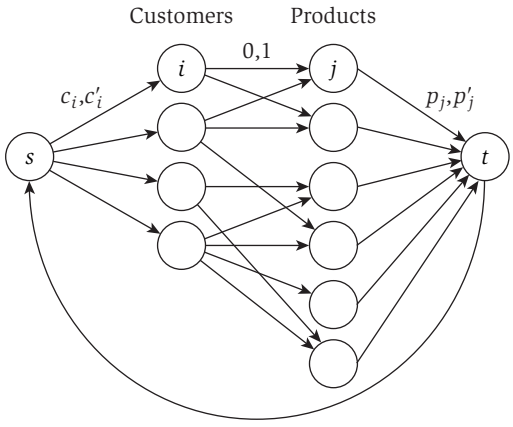

Figure 7.16 The Survey Design Problem can be reduced to the problem of finding a feasible circulation: Flow passes from customers (with capacity bounds indicating how many questions they can be asked) to products (with capacity bounds indicating how many questions should be asked about each product).

Designing the Algorithm

We will solve this problem by reducing it to a circulation problem on a flow network G' with demands and lower bounds as shown in Figure 7.16. To obtain the graph G' from G, we orient the edges of G from customers to products, add nodes s and t with edges (s, i) for each customer $i = 1, \ldots, n$, edges (j, t) for each product $j = 1, \ldots, k$, and an edge (t, s). The circulation in this network will correspond to the way in which questions are asked. The flow on the edge (s, i) is the number of products included on the questionnaire for customer i, so this edge will have a capacity of c'_i and a lower bound of c_i. The flow on the edge (j, t) will correspond to the number of customers who were asked about product j, so this edge will have a capacity of p'_j and a lower bound of p_j. Each edge (i, j) going from a customer to a product he or she bought has capacity 1, and 0 as the lower bound. The flow carried by the edge (t, s) corresponds to the overall number of questions asked. We can give this edge a capacity of $\sum_i c'_i$ and a lower bound of $\sum_i c_i$. All nodes have demand 0.

Our algorithm is simply to construct this network G' and check whether it has a feasible circulation. We now formulate a claim that establishes the correctness of this algorithm.

Analyzing the Algorithm

(7.53) *The graph G' just constructed has a feasible circulation if and only if there is a feasible way to design the survey.*

Proof. The construction above immediately suggests a way to turn a survey design into the corresponding flow. The edge (i, j) will carry one unit of flow if customer i is asked about product j in the survey, and will carry no flow otherwise. The flow on the edges (s, i) is the number of questions asked from customer i, the flow on the edge (j, t) is the number of customers who were asked about product j, and finally, the flow on edge (t, s) is the overall number of questions asked. This flow satisfies the 0 demand, that is, there is flow conservation at every node. If the survey satisfies these rules, then the corresponding flow satisfies the capacities and lower bounds.

Conversely, if the Circulation Problem is feasible, then by (7.52) there is a feasible circulation that is integer-valued, and such an integer-valued circulation naturally corresponds to a feasible survey design. Customer i will be surveyed about product j if and only if the edge (i, j) carries a unit of flow. ■

7.9 Airline Scheduling

The computational problems faced by the nation's large airline carriers are almost too complex to even imagine. They have to produce schedules for thousands of routes each day that are efficient in terms of equipment usage, crew allocation, customer satisfaction, and a host of other factors—all in the face of unpredictable issues like weather and breakdowns. It's not surprising that they're among the largest consumers of high-powered algorithmic techniques.

Covering these computational problems in any realistic level of detail would take us much too far afield. Instead, we'll discuss a "toy" problem that captures, in a very clean way, some of the resource allocation issues that arise in a context such as this. And, as is common in this book, the toy problem will be much more useful for our purposes than the "real" problem, for the solution to the toy problem involves a very general technique that can be applied in a wide range of situations.

The Problem

Suppose you're in charge of managing a fleet of airplanes and you'd like to create a flight schedule for them. Here's a very simple model for this. Your market research has identified a set of m particular flight segments that would be very lucrative if you could serve them; flight segment j is specified by four parameters: its origin airport, its destination airport, its departure time, and its arrival time. Figure 7.17(a) shows a simple example, consisting of six flight segments you'd like to serve with your planes over the course of a single day:

(1) Boston (depart 6 A.M.) – Washington DC (arrive 7 A.M.)
(2) Philadelphia (depart 7 A.M.) – Pittsburgh (arrive 8 A.M.)

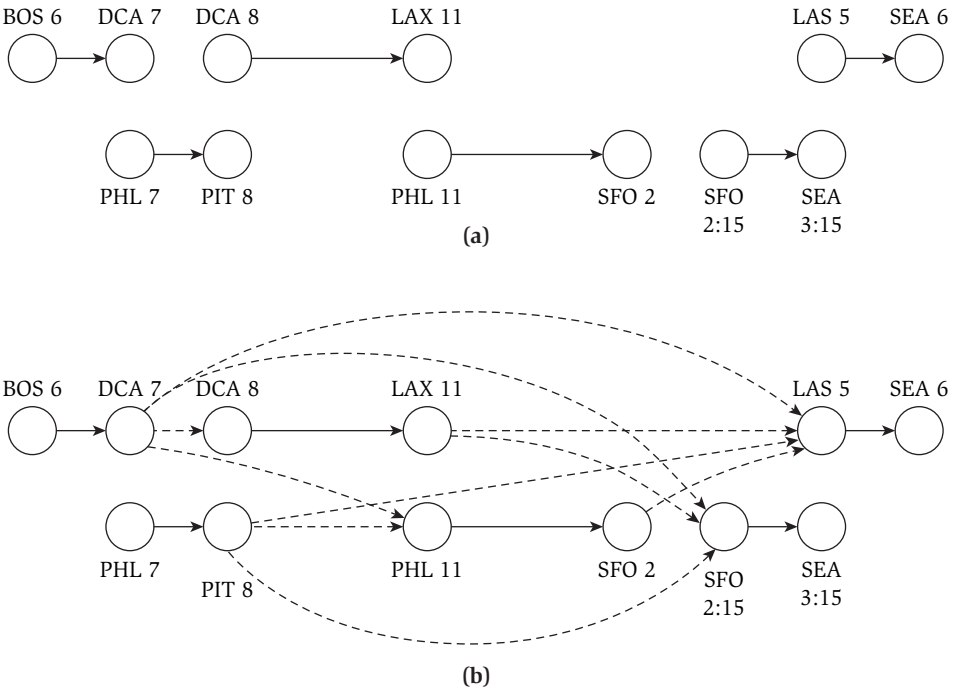

Figure 7.17 (a) A small instance of our simple Airline Scheduling Problem. (b) An expanded graph showing which flights are reachable from which others.

(3) Washington DC (depart 8 A.M.) – Los Angeles (arrive 11 A.M.)
(4) Philadelphia (depart 11 A.M.) – San Francisco (arrive 2 P.M.)
(5) San Francisco (depart 2:15 P.M.) – Seattle (arrive 3:15 P.M.)
(6) Las Vegas (depart 5 P.M.) – Seattle (arrive 6 P.M.)

Note that each segment includes the times you want the flight to serve as well as the airports.

It is possible to use a single plane for a flight segment i, and then later for a flight segment j, provided that

(a) the destination of i is the same as the origin of j, and there's enough time to perform maintenance on the plane in between; or

(b) you can add a flight segment in between that gets the plane from the destination of i to the origin of j with adequate time in between.

For example, assuming an hour for intermediate maintenance time, you could use a single plane for flights (1), (3), and (6) by having the plane sit in Washington, DC, between flights (1) and (3), and then inserting the flight

Los Angeles (depart 12 noon) – Las Vegas (1 P.M.)

in between flights (3) and (6).

Formulating the Problem We can model this situation in a very general way as follows, abstracting away from specific rules about maintenance times and intermediate flight segments: We will simply say that flight j is *reachable* from flight i if it is possible to use the same plane for flight i, and then later for flight j as well. So under our specific rules (a) and (b) above, we can easily determine for each pair i, j whether flight j is reachable from flight i. (Of course, one can easily imagine more complex rules for reachability. For example, the length of maintenance time needed in (a) might depend on the airport; or in (b) we might require that the flight segment you insert be sufficiently profitable on its own.) But the point is that we can handle any set of rules with our definition: The input to the problem will include not just the flight segments, but also a specification of the pairs (i, j) for which a later flight j is reachable from an earlier flight i. These pairs can form an arbitrary directed acyclic graph.

The goal in this problem is to determine whether it's possible to serve all m flights on your original list, using at most k planes total. In order to do this, you need to find a way of efficiently reusing planes for multiple flights.

For example, let's go back to the instance in Figure 7.17 and assume we have $k = 2$ planes. If we use one of the planes for flights (1), (3), and (6) as proposed above, we wouldn't be able to serve all of flights (2), (4), and (5) with the other (since there wouldn't be enough maintenance time in San Francisco between flights (4) and (5)). However, there is a way to serve all six flights using two planes, via a different solution: One plane serves flights (1), (3), and (5) (splicing in an LAX–SFO flight), while the other serves (2), (4), and (6) (splicing in PIT–PHL and SFO–LAS).

Designing the Algorithm

We now discuss an efficient algorithm that can solve arbitrary instances of the Airline Scheduling Problem, based on network flow. We will see that flow techniques adapt very naturally to this problem.

The solution is based on the following idea. Units of flow will correspond to airplanes. We will have an edge for each flight, and upper and lower capacity bounds of 1 on these edges to require that exactly one unit of flow crosses this edge. In other words, each flight must be served by one of the planes. If (u_i, v_i) is the edge representing flight i, and (u_j, v_j) is the edge representing flight j, and flight j is reachable from flight i, then we will have an edge from v_i to u_j

with capacity 1; in this way, a unit of flow can traverse (u_i, v_i) and then move directly to (u_j, v_j). Such a construction of edges is shown in Figure 7.17(b).

We extend this to a flow network by including a source and sink; we now give the full construction in detail. The node set of the underlying graph G is defined as follows.

- For each flight i, the graph G will have the two nodes u_i and v_i.
- G will also have a distinct source node s and sink node t.

The edge set of G is defined as follows.

- For each i, there is an edge (u_i, v_i) with a lower bound of 1 and a capacity of 1. *(Each flight on the list must be served.)*
- For each i and j so that flight j is reachable from flight i, there is an edge (v_i, u_j) with a lower bound of 0 and a capacity of 1. *(The same plane can perform flights i and j.)*
- For each i, there is an edge (s, u_i) with a lower bound of 0 and a capacity of 1. *(Any plane can begin the day with flight i.)*
- For each j, there is an edge (v_j, t) with a lower bound of 0 and a capacity of 1. *(Any plane can end the day with flight j.)*
- There is an edge (s, t) with lower bound 0 and capacity k. *(If we have extra planes, we don't need to use them for any of the flights.)*

Finally, the node s will have a demand of $-k$, and the node t will have a demand of k. All other nodes will have a demand of 0.

Our algorithm is to construct the network G and search for a feasible circulation in it. We now prove the correctness of this algorithm.

Analyzing the Algorithm

(7.54) *There is a way to perform all flights using at most k planes if and only if there is a feasible circulation in the network G.*

Proof. First, suppose there is a way to perform all flights using $k' \leq k$ planes. The set of flights performed by each individual plane defines a path P in the network G, and we send one unit of flow on each such path P. To satisfy the full demands at s and t, we send $k - k'$ units of flow on the edge (s, t). The resulting circulation satisfies all demand, capacity, and lower bound conditions.

Conversely, consider a feasible circulation in the network G. By (7.52), we know that there is a feasible circulation with integer flow values. Suppose that k' units of flow are sent on edges other than (s, t). Since all other edges have a capacity bound of 1, and the circulation is integer-valued, each such edge that carries flow has exactly one unit of flow on it.

We now convert this to a schedule using the same kind of construction we saw in the proof of (7.42), where we converted a flow to a collection of paths. In fact, the situation is easier here since the graph has no cycles. Consider an edge (s, u_i) that carries one unit of flow. It follows by conservation that (u_i, v_i) carries one unit of flow, and that there is a unique edge out of v_i that carries one unit of flow. If we continue in this way, we construct a path P from s to t, so that each edge on this path carries one unit of flow. We can apply this construction to each edge of the form (s, u_j) carrying one unit of flow; in this way, we produce k' paths from s to t, each consisting of edges that carry one unit of flow. Now, for each path P we create in this way, we can assign a single plane to perform all the flights contained in this path. ■

Extensions: Modeling Other Aspects of the Problem

Airline scheduling consumes countless hours of CPU time in real life. We mentioned at the beginning, however, that our formulation here is really a toy problem; it ignores several obvious factors that would have to be taken into account in these applications. First of all, it ignores the fact that a given plane can only fly a certain number of hours before it needs to be temporarily taken out of service for more significant maintenance. Second, we are making up an optimal schedule for a single day (or at least for a single span of time) as though there were no yesterday or tomorrow; in fact we also need the planes to be optimally positioned for the start of day $N + 1$ at the end of day N. Third, all these planes need to be staffed by flight crews, and while crews are also reused across multiple flights, a whole different set of constraints operates here, since human beings and airplanes experience fatigue at different rates. And these issues don't even begin to cover the fact that serving any particular flight segment is not a hard constraint; rather, the real goal is to optimize revenue, and so we can pick and choose among many possible flights to include in our schedule (not to mention designing a good fare structure for passengers) in order to achieve this goal.

Ultimately, the message is probably this: Flow techniques are useful for solving problems of this type, and they are genuinely used in practice. Indeed, our solution above is a general approach to the efficient reuse of a limited set of resources in many settings. At the same time, running an airline efficiently in real life is a very difficult problem.

7.10 Image Segmentation

A central problem in image processing is the *segmentation* of an image into various coherent regions. For example, you may have an image representing a picture of three people standing in front of a complex background scene. A

natural but difficult goal is to identify each of the three people as coherent objects in the scene.

The Problem

One of the most basic problems to be considered along these lines is that of foreground/background segmentation: We wish to label each pixel in an image as belonging to either the foreground of the scene or the background. It turns out that a very natural model here leads to a problem that can be solved efficiently by a minimum cut computation.

Let V be the set of *pixels* in the underlying image that we're analyzing. We will declare certain pairs of pixels to be *neighbors*, and use E to denote the set of all pairs of neighboring pixels. In this way, we obtain an *undirected* graph $G = (V, E)$. We will be deliberately vague on what exactly we mean by a "pixel," or what we mean by the "neighbor" relation. In fact, any graph G will yield an efficiently solvable problem, so we are free to define these notions in any way that we want. Of course, it is natural to picture the pixels as constituting a grid of dots, and the neighbors of a pixel to be those that are directly adjacent to it in this grid, as shown in Figure 7.18(a).

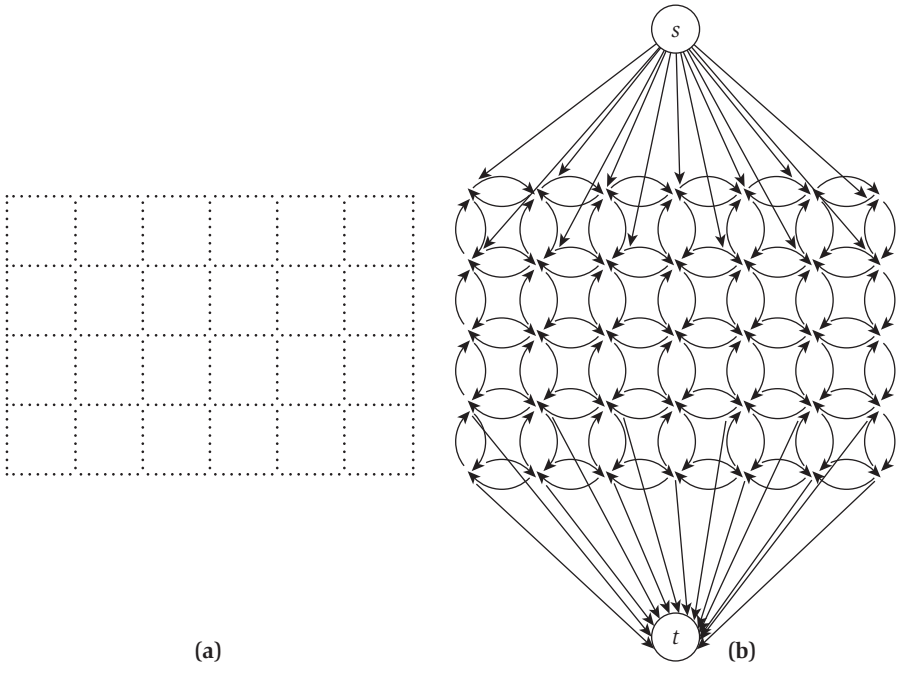

Figure 7.18 (a) A pixel graph. (b) A sketch of the corresponding flow graph. Not all edges from the source or to the sink are drawn.

For each pixel i, we have a *likelihood* a_i that it belongs to the foreground, and a *likelihood* b_i that it belongs to the background. For our purposes, we will assume that these likelihood values are arbitrary nonnegative numbers provided as part of the problem, and that they specify how desirable it is to have pixel i in the background or foreground. Beyond this, it is not crucial precisely what physical properties of the image they are measuring, or how they were determined.

In isolation, we would want to label pixel i as belonging to the foreground if $a_i > b_i$, and to the background otherwise. However, decisions that we make about the neighbors of i should affect our decision about i. If many of i's neighbors are labeled "background," for example, we should be more inclined to label i as "background" too; this makes the labeling "smoother" by minimizing the amount of foreground/background boundary. Thus, for each pair (i, j) of neighboring pixels, there is a *separation penalty* $p_{ij} \geq 0$ for placing one of i or j in the foreground and the other in the background.

We can now specify our *Segmentation Problem* precisely, in terms of the likelihood and separation parameters: It is to find a partition of the set of pixels into sets A and B (foreground and background, respectively) so as to maximize

$$q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}.$$

Thus we are rewarded for having high likelihood values and penalized for having neighboring pairs (i, j) with one pixel in A and the other in B. The problem, then, is to compute an *optimal labeling*—a partition (A, B) that maximizes $q(A, B)$.

Designing and Analyzing the Algorithm

We notice right away that there is clearly a resemblance between the minimum-cut problem and the problem of finding an optimal labeling. However, there are a few significant differences. First, we are seeking to maximize an objective function rather than minimizing one. Second, there is no source and sink in the labeling problem; and, moreover, we need to deal with values a_i and b_i on the nodes. Third, we have an undirected graph G, whereas for the minimum-cut problem we want to work with a directed graph. Let's address these problems in order.

We deal with the fact that our Segmentation Problem is a maximization problem through the following observation. Let $Q = \sum_i (a_i + b_i)$. The sum $\sum_{i \in A} a_i + \sum_{j \in B} b_j$ is the same as the sum $Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j$, so we can write

$$q(A, B) = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}.$$

Thus we see that the maximization of $q(A, B)$ is the same problem as the minimization of the quantity

$$q'(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}.$$

As for the missing source and the sink, we work by analogy with our constructions in previous sections: We create a new "super-source" s to represent the foreground, and a new "super-sink" t to represent the background. This also gives us a way to deal with the values a_i and b_i that reside at the nodes (whereas minimum cuts can only handle numbers associated with edges). Specifically, we will attach each of s and t to every pixel, and use a_i and b_i to define appropriate capacities on the edges between pixel i and the source and sink respectively.

Finally, to take care of the undirected edges, we model each neighboring pair (i, j) with *two* directed edges, (i, j) and (j, i), as we did in the undirected Disjoint Paths Problem. We will see that this works very well here too, since in any s-t cut, at most one of these two oppositely directed edges can cross from the s-side to the t-side of the cut (for if one does, then the other must go from the t-side to the s-side).

Specifically, we define the following flow network $G' = (V', E')$ shown in Figure 7.18(b). The node set V' consists of the set V of pixels, together with two additional nodes s and t. For each neighboring pair of pixels i and j, we add directed edges (i, j) and (j, i), each with capacity p_{ij}. For each pixel i, we add an edge (s, i) with capacity a_i and an edge (i, t) with capacity b_i.

Now, an s-t cut (A, B) corresponds to a partition of the pixels into sets A and B. Let's consider how the capacity of the cut $c(A, B)$ relates to the quantity $q'(A, B)$ that we are trying to minimize. We can group the edges that cross the cut (A, B) into three natural categories.

- Edges (s, j), where $j \in B$; this edge contributes a_j to the capacity of the cut.
- Edges (i, t), where $i \in A$; this edge contributes b_i to the capacity of the cut.
- Edges (i, j) where $i \in A$ and $j \in B$; this edge contributes p_{ij} to the capacity of the cut.

Figure 7.19 illustrates what each of these three kinds of edges looks like relative to a cut, on an example with four pixels.

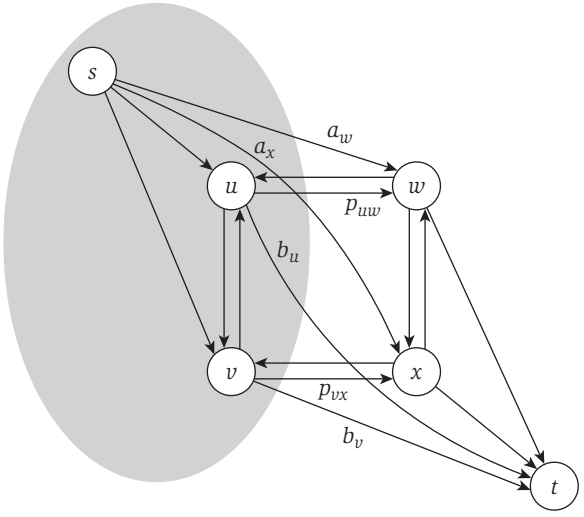

Figure 7.19 An s-t cut on a graph constructed from four pixels. Note how the three types of terms in the expression for $q'(A, B)$ are captured by the cut.

If we add up the contributions of these three kinds of edges, we get

$$c(A,B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}|=1}} p_{ij}$$
$$= q'(A,B).$$

So everything fits together perfectly. The flow network is set up so that the capacity of the cut (A, B) exactly measures the quantity $q'(A, B)$: The three kinds of edges crossing the cut (A, B), as we have just defined them (edges from the source, edges to the sink, and edges involving neither the source nor the sink), correspond to the three kinds of terms in the expression for $q'(A, B)$.

Thus, if we want to minimize $q'(A, B)$ (since we have argued earlier that this is equivalent to maximizing $q(A, B)$), we just have to find a cut of minimum capacity. And this latter problem, of course, is something that we know how to solve efficiently.

Thus, through solving this minimum-cut problem, we have an optimal algorithm in our model of foreground/background segmentation.

(7.55) *The solution to the Segmentation Problem can be obtained by a minimum-cut algorithm in the graph G' constructed above. For a minimum cut (A', B'), the partition (A, B) obtained by deleting s^* and t^* maximizes the segmentation value $q(A, B)$.*

7.11 Project Selection

Large (and small) companies are constantly faced with a balancing act between projects that can yield revenue, and the expenses needed for activities that can support these projects. Suppose, for example, that the telecommunications giant CluNet is assessing the pros and cons of a project to offer some new type of high-speed access service to residential customers. Marketing research shows that the service will yield a good amount of revenue, but it must be weighed against some costly preliminary projects that would be needed in order to make this service possible: increasing the fiber-optic capacity in the core of their network, and buying a newer generation of high-speed routers.

What makes these types of decisions particularly tricky is that they interact in complex ways: in isolation, the revenue from the high-speed access service might not be enough to justify modernizing the routers; *however*, once the company has modernized the routers, they'll also be in a position to pursue a lucrative additional project with their corporate customers; and maybe this additional project will tip the balance. And these interactions chain together: the corporate project actually would require another expense, but this in turn would enable two other lucrative projects—and so forth. In the end, the question is: Which projects should be pursued, and which should be passed up? It's a basic issue of balancing costs incurred with profitable opportunities that are made possible.

The Problem

Here's a very general framework for modeling a set of decisions such as this. There is an underlying set P of *projects*, and each project $i \in P$ has an associated *revenue* p_i, which can either be positive or negative. (In other words, each of the lucrative opportunities and costly infrastructure-building steps in our example above will be referred to as a separate project.) Certain projects are prerequisites for other projects, and we model this by an underlying directed acyclic graph $G = (P, E)$. The nodes of G are the projects, and there is an edge (i, j) to indicate that project i can only be selected if project j is selected as well. Note that a project i can have many prerequisites, and there can be many projects that have project j as one of their prerequisites. A set of projects $A \subseteq P$ is *feasible* if the prerequisite of every project in A also belongs to A: for each $i \in A$, and each edge $(i, j) \in E$, we also have $j \in A$. We will refer to requirements of this form as *precedence constraints*. The profit of a set of projects is defined to be

$$\texttt{profit}(A) = \sum_{i \in A} p_i.$$

The *Project Selection Problem* is to select a feasible set of projects with maximum profit.

This problem also became a hot topic of study in the mining literature, starting in the early 1960s; here it was called the *Open-Pit Mining Problem*.[3] Open-pit mining is a surface mining operation in which blocks of earth are extracted from the surface to retrieve the ore contained in them. Before the mining operation begins, the entire area is divided into a set P of *blocks*, and the net value p_i of each block is estimated: This is the value of the ore minus the processing costs, for this block considered in isolation. Some of these net values will be positive, others negative. The full set of blocks has precedence constraints that essentially prevent blocks from being extracted before others on top of them are extracted. The Open-Pit Mining Problem is to determine the most profitable set of blocks to extract, subject to the precedence constraints. This problem falls into the framework of project selection—each block corresponds to a separate project.

Designing the Algorithm

Here we will show that the Project Selection Problem can be solved by reducing it to a minimum-cut computation on an extended graph G', defined analogously to the graph we used in Section 7.10 for image segmentation. The idea is to construct G' from G in such a way that the source side of a minimum cut in G' will correspond to an optimal set of projects to select.

To form the graph G', we add a new source s and a new sink t to the graph G as shown in Figure 7.20. For each node $i \in P$ with $p_i > 0$, we add an edge (s, i) with capacity p_i. For each node $i \in P$ with $p_i < 0$, we add an edge (i, t) with capacity $-p_i$. We will set the capacities on the edges in G later. However, we can already see that the capacity of the cut $(\{s\}, P \cup \{t\})$ is $C = \sum_{i \in P: p_i > 0} p_i$, so the maximum-flow value in this network is at most C.

We want to ensure that if (A', B') is a minimum cut in this graph, then $A = A' - \{s\}$ obeys the precedence constraints; that is, if the node $i \in A$ has an edge $(i, j) \in E$, then we must have $j \in A$. The conceptually cleanest way to ensure this is to give each of the edges in G capacity of ∞. We haven't previously formalized what an infinite capacity would mean, but there is no problem in doing this: it is simply an edge for which the capacity condition imposes no upper bound at all. The algorithms of the previous sections, as well as the Max-Flow Min-Cut Theorem, carry over to handle infinite capacities. However, we can also avoid bringing in the notion of infinite capacities by

[3] In contrast to the field of data mining, which has motivated several of the problems we considered earlier, we're talking here about actual mining, where you dig things out of the ground.

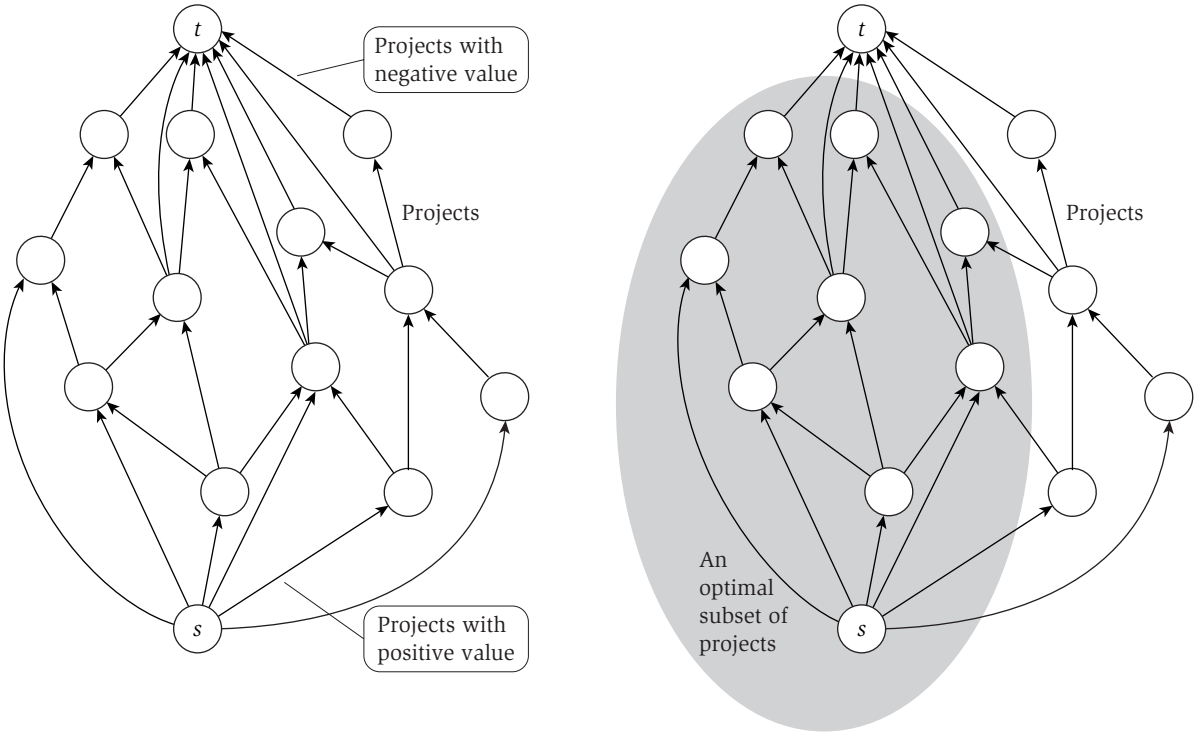

Figure 7.20 The flow graph used to solve the Project Selection Problem. A possible minimum-capacity cut is shown on the right.

simply assigning each of these edges a capacity that is "effectively infinite." In our context, giving each of these edges a capacity of $C + 1$ would accomplish this: The maximum possible flow value in G' is at most C, and so no minimum cut can contain an edge with capacity above C. In the description below, it will not matter which of these options we choose.

We can now state the algorithm: We compute a minimum cut (A', B') in G', and we declare $A' - \{s\}$ to be the optimal set of projects. We now turn to proving that this algorithm indeed gives the optimal solution.

Analyzing the Algorithm

First consider a set of projects A that satisfies the precedence constraints. Let $A' = A \cup \{s\}$ and $B' = (P - A) \cup \{t\}$, and consider the s-t cut (A', B'). If the set A satisfies the precedence constraints, then no edge $(i, j) \in E$ crosses this cut, as shown in Figure 7.20. The capacity of the cut can be expressed as follows.

(7.56) *The capacity of the cut (A', B'), as defined from a project set A satisfying the precedence constraints, is $c(A', B') = C - \sum_{i \in A} p_i$.*

Proof. Edges of G' can be divided into three categories: those corresponding to the edge set E of G, those leaving the source s, and those entering the sink t. Because A satisfies the precedence constraints, the edges in E do not cross the cut (A', B'), and hence do not contribute to its capacity. The edges entering the sink t contribute

$$\sum_{i \in A \text{ and } p_i < 0} -p_i$$

to the capacity of the cut, and the edges leaving the source s contribute

$$\sum_{i \notin A \text{ and } p_i > 0} p_i.$$

Using the definition of C, we can rewrite this latter quantity as $C - \sum_{i \in A \text{ and } p_i > 0} p_i$. The capacity of the cut (A', B') is the sum of these two terms, which is

$$\sum_{i \in A \text{ and } p_i < 0} (-p_i) + \left(C - \sum_{i \in A \text{ and } p_i > 0} p_i \right) = C - \sum_{i \in A} p_i,$$

as claimed. ■

Next, recall that edges of G have capacity more than $C = \sum_{i \in P : p_i > 0} p_i$, and so these edges cannot cross a cut of capacity at most C. This implies that such cuts define feasible sets of projects.

(7.57) *If (A', B') is a cut with capacity at most C, then the set $A = A' - \{s\}$ satisfies the precedence constraints.*

Now we can prove the main goal of our construction, that the minimum cut in G' determines the optimum set of projects. Putting the previous two claims together, we see that the cuts (A', B') of capacity at most C are in one-to-one correspondence with feasible sets of project $A = A' - \{s\}$. The capacity of such a cut (A', B') is

$$c(A', B') = C - \texttt{profit}(A).$$

The capacity value C is a constant, independent of the cut (A', B'), so the cut with minimum capacity corresponds to the set of projects A with maximum profit. We have therefore proved the following.

(7.58) *If (A', B') is a minimum cut in G' then the set $A = A' - \{s\}$ is an optimum solution to the Project Selection Problem.*

7.12 Baseball Elimination

Over on the radio side the producer's saying, "See that thing in the paper last week about Einstein? . . . Some reporter asked him to figure out the mathematics of the pennant race. You know, one team wins so many of their remaining games, the other teams win this number or that number. What are the myriad possibilities? Who's got the edge?"

"The hell does he know?"

"Apparently not much. He picked the Dodgers to eliminate the Giants last Friday."

—Don DeLillo, *Underworld*

The Problem

Suppose you're a reporter for the *Algorithmic Sporting News*, and the following situation arises late one September. There are four baseball teams trying to finish in first place in the American League Eastern Division; let's call them New York, Baltimore, Toronto, and Boston. Currently, each team has the following number of wins:

New York: 92, Baltimore: 91, Toronto: 91, Boston: 90.

There are five games left in the season: These consist of all possible pairings of the four teams above, except for New York and Boston.

The question is: Can Boston finish with at least as many wins as every other team in the division (that is, finish in first place, possibly in a tie)?

If you think about it, you realize that the answer is no. One argument is the following. Clearly, Boston must win both its remaining games and New York must lose both its remaining games. But this means that Baltimore and Toronto will both beat New York; so then the winner of the Baltimore-Toronto game will end up with the most wins.

Here's an argument that avoids this kind of cases analysis. Boston can finish with at most 92 wins. Cumulatively, the other three teams have 274 wins currently, and their three games against each other will produce exactly three more wins, for a final total of 277. But 277 wins over three teams means that one of them must have ended up with more than 92 wins.

So now you might start wondering: (i) Is there an efficient algorithm to determine whether a team has been eliminated from first place? And (ii) whenever a team has been eliminated from first place, is there an "averaging" argument like this that proves it?

In more concrete notation, suppose we have a set S of teams, and for each $x \in S$, its current number of wins is w_x. Also, for two teams $x, y \in S$, they still

have to play g_{xy} games against one another. Finally, we are given a specific team z.

We will use maximum-flow techniques to achieve the following two things. First, we give an efficient algorithm to decide whether z has been eliminated from first place—or, to put it in positive terms, whether it is possible to choose outcomes for all the remaining games in such a way that the team z ends with at least as many wins as every other team in S. Second, we prove the following clean characterization theorem for baseball elimination—essentially, that there is always a short "proof" when a team has been eliminated.

(7.59) *Suppose that team z has indeed been eliminated. Then there exists a "proof" of this fact of the following form:*

- *z can finish with at most m wins.*
- *There is a set of teams $T \subseteq S$ so that*

$$\sum_{x \in T} w_x + \sum_{x,y \in T} g_{xy} > m|T|.$$

(And hence one of the teams in T must end with strictly more than m wins.)

As a second, more complex illustration of how the averaging argument in (7.59) works, consider the following example. Suppose we have the same four teams as before, but now the current number of wins is

New York: 90, Baltimore: 88, Toronto: 87, Boston: 79.

The remaining games are as follows. Boston still has four games against each of the other three teams. Baltimore has one more game against each of New York and Toronto. And finally, New York and Toronto still have six games left to play against each other. Clearly, things don't look good for Boston, but is it actually eliminated?

The answer is yes; Boston has been eliminated. To see this, first note that Boston can end with at most 91 wins; and now consider the set of teams $T = \{\text{New York}, \text{Toronto}\}$. Together New York and Toronto already have 177 wins; their six remaining games will result in a total of 183; and $\frac{183}{2} > 91$. This means that one of them must end up with more than 91 wins, and so Boston can't finish in first. Interestingly, in this instance the set of all three teams ahead of Boston cannot constitute a similar proof: All three teams taken togeher have a total of 265 wins with 8 games left among them; this is a total of 273, and $\frac{273}{3} = 91$ — not enough by itself to prove that Boston couldn't end up in a multi-way tie for first. So it's crucial for the averaging argument that we choose the set T consisting just of New York and Toronto, and omit Baltimore.

Designing and Analyzing the Algorithm

We begin by constructing a flow network that provides an efficient algorithm for determining whether z has been eliminated. Then, by examining the minimum cut in this network, we will prove (7.59).

Clearly, if there's any way for z to end up in first place, we should have z win all its remaining games. Let's suppose that this leaves it with m wins. We now want to carefully allocate the wins from all remaining games so that no other team ends with more than m wins. Allocating wins in this way can be solved by a maximum-flow computation, via the following basic idea. We have a source s from which all wins emanate. The i^{th} win can pass through one of the two teams involved in the i^{th} game. We then impose a capacity constraint saying that at most $m - w_x$ wins can pass through team x.

More concretely, we construct the following flow network G, as shown in Figure 7.21. First, let $S' = S - \{z\}$, and let $g^* = \sum_{x,y \in S'} g_{xy}$—the total number of games left between all pairs of teams in S'. We include nodes s and t, a node v_x for each team $x \in S'$, and a node u_{xy} for each pair of teams $x, y \in S'$ with a nonzero number of games left to play against each other. We have the following edges.

- Edges (s, u_{xy}) (*wins emanate from s*);
- Edges (u_{xy}, v_x) and (u_{xy}, v_y) (*only x or y can win a game that they play against each other*); and
- Edges (v_x, t) (*wins are absorbed at t*).

Let's consider what capacities we want to place on these edges. We want g_{xy} wins to flow from s to u_{xy} at saturation, so we give (s, u_{xy}) a capacity of g_{xy}. We want to ensure that team x cannot win more than $m - w_x$ games, so we

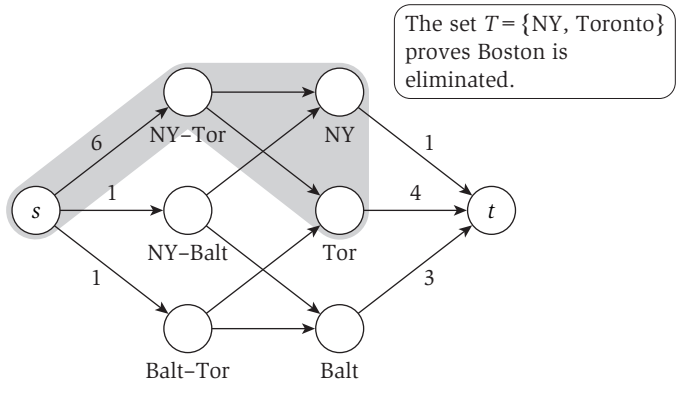

Figure 7.21 The flow network for the second example. As the minimum cut indicates, there is no flow of value g^*, and so Boston has been eliminated.

give the edge (v_x, t) a capacity of $m - w_x$. Finally, an edge of the form (u_{xy}, v_y) should have *at least* g_{xy} units of capacity, so that it has the ability to transport all the wins from u_{xy} on to v_x; in fact, our analysis will be the cleanest if we give it *infinite* capacity. (We note that the construction still works even if this edge is given only g_{xy} units of capacity, but the proof of (7.59) will become a little more complicated.)

Now, if there is a flow of value g^*, then it is possible for the outcomes of all remaining games to yield a situation where no team has more than m wins; and hence, if team z wins all its remaining games, it can still achieve at least a tie for first place. Conversely, if there are outcomes for the remaining games in which z achieves at least a tie, we can use these outcomes to define a flow of value g^*. For example, in Figure 7.21, which is based on our second example, the indicated cut shows that the maximum flow has value at most 7, whereas $g^* = 6 + 1 + 1 = 8$.

In summary, we have shown

(7.60) *Team z has been eliminated if and only if the maximum flow in G has value strictly less than g^*. Thus we can test in polynomial time whether z has been eliminated.*

Characterizing When a Team Has Been Eliminated

Our network flow construction can also be used to prove (7.59). The idea is that the Max-Flow Min-Cut Theorem gives a nice "if and only if" characterization for the existence of flow, and if we interpret this characterization in terms of our application, we get the comparably nice characterization here. This illustrates a general way in which one can generate characterization theorems for problems that are reducible to network flow.

Proof of (7.59). Suppose that z has been eliminated from first place. Then the maximum s-t flow in G has value $g' < g^*$; so there is an s-t cut (A, B) of capacity g', and (A, B) is a minimum cut. Let T be the set of teams x for which $v_x \in A$. We will now prove that T can be used in the "averaging argument" in (7.59).

First, consider the node u_{xy}, and suppose one of x or y is not in T, but $u_{xy} \in A$. Then the edge (u_{xy}, v_x) would cross from A into B, and hence the cut (A, B) would have infinite capacity. This contradicts the assumption that (A, B) is a minimum cut of capacity less than g^*. So if one of x or y is not in T, then $u_{xy} \in B$. On the other hand, suppose both x and y belong to T, but $u_{xy} \in B$. Consider the cut (A', B') that we would obtain by adding u_{xy} to the set A and deleting it from the set B. The capacity of (A', B') is simply the capacity of (A, B), minus the capacity g_{xy} of the edge (s, u_{xy})—for this edge (s, u_{xy}) used

to cross from A to B, and now it does not cross from A' to B'. But since $g_{xy} > 0$, this means that (A', B') has smaller capacity than (A, B), again contradicting our assumption that (A, B) is a minimum cut. So, if both x and y belong to T, then $u_{xy} \in A$.

Thus we have established the following conclusion, based on the fact that (A, B) is a minimum cut: $u_{xy} \in A$ if and only if both $x, y \in T$.

Now we just need to work out the minimum-cut capacity $c(A, B)$ in terms of its constituent edge capacities. By the conclusion in the previous paragraph, we know that edges crossing from A to B have one of the following two forms:

- edges of the form (v_x, t), where $x \in T$, and
- edges of the form (s, u_{xy}), where at least one of x or y does not belong to T (in other words, $\{x, y\} \not\subset T$).

Thus we have

$$\begin{aligned} c(A, B) &= \sum_{x \in T} (m - w_x) + \sum_{\{x,y\} \not\subset T} g_{xy} \\ &= m|T| - \sum_{x \in T} w_x + (g^* - \sum_{x,y \in T} g_{xy}). \end{aligned}$$

Since we know that $c(A, B) = g' < g^*$, this last inequality implies

$$m|T| - \sum_{x \in T} w_x - \sum_{x,y \in T} g_{xy} < 0,$$

and hence

$$\sum_{x \in T} w_x + \sum_{x,y \in T} g_{xy} > m|T|. \quad \blacksquare$$

For example, applying the argument in the proof of (7.59) to the instance in Figure 7.21, we see that the nodes for New York and Toronto are on the source side of the minimum cut, and, as we saw earlier, these two teams indeed constitute a proof that Boston has been eliminated.

* 7.13 A Further Direction: Adding Costs to the Matching Problem

Let's go back to the first problem we discussed in this chapter, Bipartite Matching. Perfect matchings in a bipartite graph formed a way to model the problem of pairing one kind of object with another—jobs with machines, for example. But in many settings, there are a large number of possible perfect matchings on the same set of objects, and we'd like a way to express the idea that some perfect matchings may be "better" than others.

The Problem

A natural way to formulate a problem based on this notion is to introduce *costs*. It may be that we incur a certain cost to perform a given job on a given machine, and we'd like to match jobs with machines in a way that minimizes the total cost. Or there may be n fire trucks that must be sent to n distinct houses; each house is at a given distance from each fire station, and we'd like a matching that minimizes the average distance each truck drives to its associated house. In short, it is very useful to have an algorithm that finds a perfect matching *of minimum total cost*.

Formally, we consider a bipartite graph $G = (V, E)$ whose node set, as usual, is partitioned as $V = X \cup Y$ so that every edge $e \in E$ has one end in X and the other end in Y. Furthermore, each edge e has a nonnegative cost c_e. For a matching M, we say that the cost of the matching is the total cost of all edges in M, that is, $cost(M) = \sum_{e \in M} c_e$. The *Minimum-Cost Perfect Matching Problem* assumes that $|X| = |Y| = n$, and the goal is to find a perfect matching of minimum cost.

Designing and Analyzing the Algorithm

We now describe an efficient algorithm to solve this problem, based on the idea of augmenting paths but adapted to take the costs into account. Thus, the algorithm will iteratively construct matchings using i edges, for each value of i from 1 to n. We will show that when the algorithm concludes with a matching of size n, it is a minimum-cost perfect matching. The high-level structure of the algorithm is quite simple. If we have a minimum-cost matching of size i, then we seek an augmenting path to produce a matching of size $i + 1$; and rather than looking for any augmenting path (as was sufficient in the case without costs), we use the cheapest augmenting path so that the larger matching will also have minimum cost.

Recall the construction of the residual graph used for finding augmenting paths. Let M be a matching. We add two new nodes s and t to the graph. We add edges (s, x) for all nodes $x \in X$ that are unmatched and edges (y, t) for all nodes $y \in Y$ that are unmatched. An edge $e = (x, y) \in E$ is oriented from x to y if e is not in the matching M and from y to x if $e \in M$. We will use G_M to denote this residual graph. Note that all edges going from Y to X are in the matching M, while the edges going from X to Y are not. Any directed s-t path P in the graph G_M corresponds to a matching one larger than M by swapping edges along P, that is, the edges in P from X to Y are added to M and all edges in P that go from Y to X are deleted from M. As before, we will call a path P in G_M an *augmenting path*, and we say that we *augment* the matching M using the path P.

Now we would like the resulting matching to have as small a cost as possible. To achieve this, we will search for a cheap augmenting path with respect to the following natural costs. The edges leaving s and entering t will have cost 0; an edge e oriented from X to Y will have cost c_e (as including this edge in the path means that we add the edge to M); and an edge e oriented from Y to X will have cost $-c_e$ (as including this edge in the path means that we delete the edge from M). We will use $cost(P)$ to denote the cost of a path P in G_M. The following statement summarizes this construction.

(7.61) *Let M be a matching and P be a path in G_M from s to t. Let M' be the matching obtained from M by augmenting along P. Then $|M'| = |M| + 1$ and $cost(M') = cost(M) + cost(P)$.*

Given this statement, it is natural to suggest an algorithm to find a minimum-cost perfect matching: We iteratively find minimum-cost paths in G_M, and use the paths to augment the matchings. But how can we be sure that the perfect matching we find is of minimum cost? Or even worse, is this algorithm even meaningful? We can only find minimum-cost paths if we know that the graph G_M has no negative cycles.

Analyzing Negative Cycles In fact, understanding the role of negative cycles in G_M is the key to analyzing the algorithm. First consider the case in which M is a perfect matching. Note that in this case the node s has no leaving edges, and t has no entering edges in G_M (as our matching is perfect), and hence no cycle in G_M contains s or t.

(7.62) *Let M be a perfect matching. If there is a negative-cost directed cycle C in G_M, then M is not minimum cost.*

Proof. To see this, we use the cycle C for augmentation, just the same way we used directed paths to obtain larger matchings. Augmenting M along C involves swapping edges along C in and out of M. The resulting new perfect matching M' has cost $cost(M') = cost(M) + cost(C)$; but $cost(C) < 0$, and hence M is not of minimum cost. ■

More importantly, the converse of this statement is true as well; so in fact a perfect matching M has minimum cost precisely when there is no negative cycle in G_M.

(7.63) *Let M be a perfect matching. If there are no negative-cost directed cycles C in G_M, then M is a minimum-cost perfect matching.*

Proof. Suppose the statement is not true, and let M' be a perfect matching of smaller cost. Consider the set of edges in one of M and M' but not in both.

Observe that this set of edges corresponds to a set of node-disjoint directed cycles in G_M. The cost of the set of directed cycles is exactly $cost(M') - cost(M)$. Assuming M' has smaller cost than M, it must be that at least one of these cycles has negative cost. ■

Our plan is thus to iterate through matchings of larger and larger size, maintaining the property that the graph G_M has no negative cycles in any iteration. In this way, our computation of a minimum-cost path will always be well defined; and when we terminate with a perfect matching, we can use (7.63) to conclude that it has minimum cost.

Maintaining Prices on the Nodes It will help to think about a numerical *price* $p(v)$ associated with each node v. These prices will help both in understanding how the algorithm runs, and they will also help speed up the implementation. One issue we have to deal with is to maintain the property that the graph G_M has no negative cycles in any iteration. How do we know that after an augmentation, the new residual graph still has no negative cycles? The prices will turn out to serve as a compact proof to show this.

To understand prices, it helps to keep in mind an economic interpretation of them. For this purpose, consider the following scenario. Assume that the set X represents people who need to be assigned to do a set of jobs Y. For an edge $e = (x, y)$, the cost c_e is a cost associated with having person x doing job y. Now we will think of the price $p(x)$ as an extra bonus we pay for person x to participate in this system, like a "signing bonus." With this in mind, the cost for assigning person x to do job y will become $p(x) + c_e$. On the other hand, we will think of the price $p(y)$ for nodes $y \in Y$ as a reward, or value gained by taking care of job y (no matter which person in X takes care of it). This way the "net cost" of assigning person x to do job y becomes $p(x) + c_e - p(y)$: this is the cost of hiring x for a bonus of $p(x)$, having him do job y for a cost of c_e, and then cashing in on the reward $p(y)$. We will call this the *reduced cost* of an edge $e = (x, y)$ and denote it by $c_e^p = p(x) + c_e - p(y)$. However, it is important to keep in mind that only the costs c_e are part of the problem description; the prices (bonuses and rewards) will be a way to think about our solution.

Specifically, we say that a set of numbers $\{p(v) : v \in V\}$ forms a set of *compatible prices* with respect to a matching M if

(i) for all unmatched nodes $x \in X$ we have $p(x) = 0$ (that is, people not asked to do any job do not need to be paid);

(ii) for all edges $e = (x, y)$ we have $p(x) + c_e \geq p(y)$ (that is, every edge has a nonnegative reduced cost); and

(iii) for all edges $e = (x, y) \in M$ we have $p(x) + c_e = p(y)$ (every edge used in the assignment has a reduced cost of 0).

Why are such prices useful? Intuitively, compatible prices suggest that the matching is cheap: Along the matched edges reward equals cost, while on all other edges the reward is no bigger than the cost. For a partial matching, this may not imply that the matching has the smallest possible cost for its size (it may be taking care of expensive jobs). However, we claim that if M is any matching for which there exists a set of compatible prices, then G_M has no negative cycles. For a perfect matching M, this will imply that M is of minimum cost by (7.63).

To see why G_M can have no negative cycles, we extend the definition of reduced cost to edges in the residual graph by using the same expression $c_e^p = p(v) + c_e - p(w)$ for any edge $e = (v, w)$. Observe that the definition of compatible prices implies that all edges in the residual graph G_M have nonnegative reduced costs. Now, note that for any cycle C, we have

$$cost(C) = \sum_{e \in C} c_e = \sum_{e \in C} c_e^p,$$

since all the terms on the right-hand side corresponding to prices cancel out. We know that each term on the right-hand side is nonnegative, and so clearly $cost(C)$ is nonnegative.

There is a second, algorithmic reason why it is useful to have prices on the nodes. When you have a graph with negative-cost edges but no negative cycles, you can compute shortest paths using the Bellman-Ford Algorithm in $O(mn)$ time. But if the graph in fact has no negative-cost edges, then you can use Dijkstra's Algorithm instead, which only requires time $O(m \log n)$—almost a full factor of n faster.

In our case, having the prices around allows us to compute shortest paths with respect to the nonnegative reduced costs c_e^p, arriving at an equivalent answer. Indeed, suppose we use Dijkstra's Algorithm to find the minimum cost $d_{p,M}(v)$ of a directed path from s to every node $v \in X \cup Y$ subject to the costs c_e^p. Given the minimum costs $d_{p,M}(y)$ for an unmatched node $y \in Y$, the (nonreduced) cost of the path from s to t through y is $d_{p,M}(y) + p(y)$, and so we find the minimum cost in $O(n)$ additional time. In summary, we have the following fact.

(7.64) *Let M be a matching, and p be compatible prices. We can use one run of Dijkstra's Algorithm and $O(n)$ extra time to find the minimum-cost path from s to t.*

Updating the Node Prices We took advantage of the prices to improve one iteration of the algorithm. In order to be ready for the next iteration, we need not only the minimum-cost path (to get the next matching), but also a way to produce a set of compatible prices with respect to the new matching.

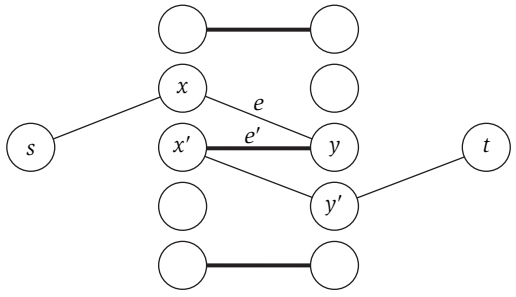

Figure 7.22 A matching M (the dark edges), and a residual graph used to increase the size of the matching.

To get some intuition on how to do this, consider an unmatched node x with respect to a matching M, and an edge $e = (x, y)$, as shown in Figure 7.22. If the new matching M' includes edge e (that is, if e is on the augmenting path we use to update the matching), then we will want to have the reduced cost of this edge to be zero. However, the prices p we used with matching M may result in a reduced cost $c_e^p > 0$ — that is, the assignment of person x to job y, in our economic interpretation, may not be viewed as cheap enough. We can arrange the zero reduced cost by either increasing the price $p(y)$ (y's reward) by c_e^p, or by decreasing the price $p(x)$ by the same amount. To keep prices nonnegative, we will increase the price $p(y)$. However, node y may be matched in the matching M to some other node x' via an edge $e' = (x', y)$, as shown in Figure 7.22. Increasing the reward $p(y)$ decreases the reduced cost of edge e' to negative, and hence the prices are no longer compatible. To keep things compatible, we can increase $p(x')$ by the same amount. However, this change might cause problems on other edges. Can we update all prices and keep the matching and the prices compatible on all edges? Surprisingly, this can be done quite simply by using the distances from s to all other nodes computed by Dijkstra's Algorithm.

(7.65) *Let M be a matching, let p be compatible prices, and let M' be a matching obtained by augmenting along the minimum-cost path from s to t. Then $p'(v) = d_{p,M}(v) + p(v)$ is a compatible set of prices for M'.*

Proof. To prove compatibility, consider first an edge $e = (x', y) \in M$. The only edge entering x' is the directed edge (y, x'), and hence $d_{p,M}(x') = d_{p,M}(y) - c_e^p$, where $c_e^p = p(y) + c_e - p(x')$, and thus we get the desired equation on such edges. Next consider edges (x, y) in $M' - M$. These edges are along the minimum-cost path from s to t, and hence they satisfy $d_{p,M}(y) = d_{p,M}(x) + c_e^p$ as desired. Finally, we get the required inequality for all other edges since all edges $e = (x, y) \notin M$ must satisfy $d_{p,M}(y) \leq d_{p,M}(x) + c_e^p$. ■

Finally, we have to consider how to initialize the algorithm, so as to get it underway. We initialize M to be the empty set, define $p(x) = 0$ for all $x \in X$, and define $p(y)$, for $y \in Y$, to be the minimum cost of an edge entering y. Note that these prices are compatible with respect to $M = \phi$.

We summarize the algorithm below.

```
Start with M equal to the empty set
Define p(x) = 0 for x ∈ X, and p(y) = min_{e into y} c_e for y ∈ Y
While M is not a perfect matching
  Find a minimum-cost s-t path P in G_M using (7.64) with prices p
  Augment along P to produce a new matching M'
  Find a set of compatible prices with respect to M' via (7.65)
Endwhile
```

The final set of compatible prices yields a proof that G_M has no negative cycles; and by (7.63), this implies that M has minimum cost.

(7.66) *The minimum-cost perfect matching can be found in the time required for n shortest-path computations with nonegative edge lengths.*

Extensions: An Economic Interpretation of the Prices

To conclude our discussion of the Minimum-Cost Perfect Matching Problem, we develop the economic interpretation of the prices a bit further. We consider the following scenario. Assume X is a set of n people each looking to buy a house, and Y is a set of n houses that they are all considering. Let $v(x, y)$ denote the value of house y to buyer x. Since each buyer wants one of the houses, one could argue that the best arrangement would be to find a perfect matching M that maximizes $\sum_{(x,y)\in M} v(x, y)$. We can find such a perfect matching by using our minimum-cost perfect matching algorithm with costs $c_e = -v(x, y)$ if $e = (x, y)$.

The question we will ask now is this: Can we convince these buyers to buy the house they are allocated? On her own, each buyer x would want to buy the house y that has maximum value $v(x, y)$ to her. How can we convince her to buy instead the house that our matching M allocated? We will use prices to change the incentives of the buyers. Suppose we set a price $P(y)$ for each house y, that is, the person buying the house y must pay $P(y)$. With these prices in mind, a buyer will be interested in buying the house with maximum net value, that is, the house y that maximizes $v(x, y) - P(y)$. We say that a

perfect matching M and house prices P are in *equilibrium* if, for all edges $(x, y) \in M$ and all other houses y', we have

$$v(x, y) - P(y) \geq v(x, y') - P(y').$$

But can we find a perfect matching and a set of prices so as to achieve this state of affairs, with every buyer ending up happy? In fact, the minimum-cost perfect matching and an associated set of compatible prices provide exactly what we're looking for.

(7.67) *Let M be a perfect matching of minimum cost, where $c_e = -v(x, y)$ for each edge $e = (x, y)$, and let p be a compatible set of prices. Then the matching M and the set of prices $\{P(y) = -p(y) : y \in Y\}$ are in equilibrium.*

Proof. Consider an edge $e = (x, y) \in M$, and let $e' = (x, y')$. Since M and p are compatible, we have $p(x) + c_e = p(y)$ and $p(x) + c_{e'} \geq p(y')$. Subtracting these two inequalities to cancel $p(x)$, and substituting the values of p and c, we get the desired inequality in the definition of equilibrium. ■

Solved Exercises

Solved Exercise 1

Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e, a designated source $s \in V$, and a designated sink $t \in V$. You are also given an integer maximum s-t flow in G, defined by a flow value f_e on each edge e.

Now suppose we pick a specific edge $e \in E$ and increase its capacity by one unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m + n)$, where m is the number of edges in G and n is the number of nodes.

Solution The point here is that $O(m + n)$ is not enough time to compute a new maximum flow from scratch, so we need to figure out how to use the flow f that we are given. Intuitively, even after we add 1 to the capacity of edge e, the flow f can't be that far from maximum; after all, we haven't changed the network very much.

In fact, it's not hard to show that the maximum flow value can go up by at most 1.

(7.68) *Consider the flow network G' obtained by adding 1 to the capacity of e. The value of the maximum flow in G' is either $v(f)$ or $v(f) + 1$.*

Proof. The value of the maximum flow in G' is at least $\nu(f)$, since f is still a feasible flow in this network. It is also integer-valued. So it is enough to show that the maximum-flow value in G' is at most $\nu(f) + 1$.

By the Max-Flow Min-Cut Theorem, there is some s-t cut (A, B) in the original flow network G of capacity $\nu(f)$. Now we ask: What is the capacity of (A, B) in the new flow network G'? All the edges crossing (A, B) have the same capacity in G' that they did in G, with the possible exception of e (in case e crosses (A, B)). But c_e only increased by 1, and so the capacity of (A, B) in the new flow network G' is at most $\nu(f) + 1$. ■

Statement (7.68) suggests a natural algorithm. Starting with the feasible flow f in G', we try to find a single augmenting path from s to t in the residual graph G'_f. This takes time $O(m + n)$. Now one of two things will happen. Either we will fail to find an augmenting path, and in this case we know that f is a maximum flow. Otherwise the augmentation succeeds, producing a flow f' of value at least $\nu(f) + 1$. In this case, we know by (7.68) that f' must be a maximum flow. So either way, we produce a maximum flow after a single augmenting path computation.

Solved Exercise 2

You are helping the medical consulting firm Doctors Without Weekends set up the work schedules of doctors in a large hospital. They've got the regular daily schedules mainly worked out. Now, however, they need to deal with all the special cases and, in particular, make sure that they have at least one doctor covering each vacation day.

Here's how this works. There are k vacation periods (e.g., the week of Christmas, the July 4th weekend, the Thanksgiving weekend, . . .), each spanning several contiguous days. Let D_j be the set of days included in the j^{th} vacation period; we will refer to the union of all these days, $\cup_j D_j$, as the set of all *vacation days.*

There are n doctors at the hospital, and doctor i has a set of vacation days S_i when he or she is available to work. (This may include certain days from a given vacation period but not others; so, for example, a doctor may be able to work the Friday, Saturday, or Sunday of Thanksgiving weekend, but not the Thursday.)

Give a polynomial-time algorithm that takes this information and determines whether it is possible to select a single doctor to work on each vacation day, subject to the following constraints.

- For a given parameter c, each doctor should be assigned to work at most c vacation days total, and only days when he or she is available.
- For each vacation period j, each doctor should be assigned to work at most one of the days in the set D_j. (In other words, although a particular doctor may work on several vacation days over the course of a year, he or she should not be assigned to work two or more days of the Thanksgiving weekend, or two or more days of the July 4th weekend, etc.)

The algorithm should either return an assignment of doctors satisfying these constraints or report (correctly) that no such assignment exists.

Solution This is a very natural setting in which to apply network flow, since at a high level we're trying to match one set (the doctors) with another set (the vacation days). The complication comes from the requirement that each doctor can work at most one day in each vacation period.

So to begin, let's see how we'd solve the problem without that requirement, in the simpler case where each doctor i has a set S_i of days when he or she can work, and each doctor should be scheduled for at most c days total. The construction is pictured in Figure 7.23(a). We have a node u_i representing each doctor attached to a node v_ℓ representing each day when he or she can

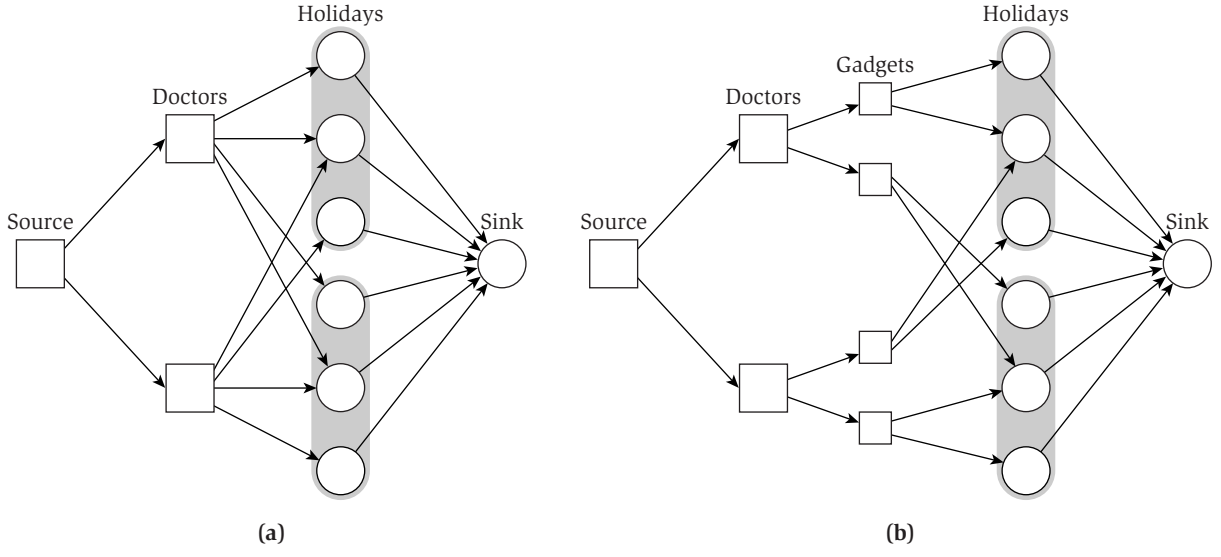

Figure 7.23 (a) Doctors are assigned to holiday days without restricting how many days in one holiday a doctor can work. (b) The flow network is expanded with "gadgets" that prevent a doctor from working more than one day from each vacation period. The shaded sets correspond to the different vacation periods.

work; this edge has a capacity of 1. We attach a super-source s to each doctor node u_i by an edge of capacity c, and we attach each day node v_ℓ to a super-sink t by an edge with upper and lower bounds of 1. This way, assigned days can "flow" through doctors to days when they can work, and the lower bounds on the edges from the days to the sink guarantee that each day is covered. Finally, suppose there are d vacation days total; we put a demand of $+d$ on the sink and $-d$ on the source, and we look for a feasible circulation. (Recall that once we've introduced lower bounds on some edges, the algorithms in the text are phrased in terms of circulations with demands, not maximum flow.)

But now we have to handle the extra requirement, that each doctor can work at most one day from each vacation period. To do this, we take each pair (i, j) consisting of a doctor i and a vacation period j, and we add a "vacation gadget" as follows. We include a new node w_{ij} with an incoming edge of capacity 1 from the doctor node u_i, and with outgoing edges of capacity 1 to each day in vacation period j when doctor i is available to work. This gadget serves to "choke off" the flow from u_i into the days associated with vacation period j, so that at most one unit of flow can go to them collectively. The construction is pictured in Figure 7.23(b). As before, we put a demand of $+d$ on the sink and $-d$ on the source, and we look for a feasible circulation. The total running time is the time to construct the graph, which is $O(nd)$, plus the time to check for a single feasible circulation in this graph.

The correctness of the algorithm is a consequence of the following claim.

(7.69) *There is a way to assign doctors to vacation days in a way that respects all constraints if and only if there is a feasible circulation in the flow network we have constructed.*

Proof. First, if there is a way to assign doctors to vacation days in a way that respects all constraints, then we can construct the following circulation. If doctor i works on day ℓ of vacation period j, then we send one unit of flow along the path s, u_i, w_{ij}, v_ℓ, t; we do this for all such (i, ℓ) pairs. Since the assignment of doctors satisfied all the constraints, the resulting circulation respects all capacities; and it sends d units of flow out of s and into t, so it meets the demands.

Conversely, suppose there is a feasible circulation. For this direction of the proof, we will show how to use the circulation to construct a schedule for all the doctors. First, by (7.52), there is a feasible circulation in which all flow values are integers. We now construct the following schedule: If the edge (w_{ij}, v_ℓ) carries a unit of flow, then we have doctor i work on day ℓ. Because of the capacities, the resulting schedule has each doctor work at most c days, at most one in each vacation period, and each day is covered by one doctor.

■

Exercises

1. **(a)** List all the minimum *s*-*t* cuts in the flow network pictured in Figure 7.24. The capacity of each edge appears as a label next to the edge.

(b) What is the minimum capacity of an *s*-*t* cut in the flow network in Figure 7.25? Again, the capacity of each edge appears as a label next to the edge.

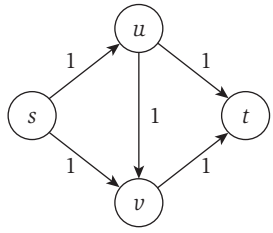

Figure 7.24 What are the minimum *s*-*t* cuts in this flow network?

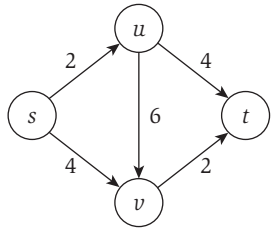

Figure 7.25 What is the minimum capacity of an *s*-*t* cut in this flow network?

2. Figure 7.26 shows a flow network on which an *s*-*t* flow has been computed. The capacity of each edge appears as a label next to the edge, and the numbers in boxes give the amount of flow sent on each edge. (Edges without boxed numbers—specifically, the four edges of capacity 3—have no flow being sent on them.)

(a) What is the value of this flow? Is this a maximum (s,t) flow in this graph?

(b) Find a minimum *s*-*t* cut in the flow network pictured in Figure 7.26, and also say what its capacity is.

3. Figure 7.27 shows a flow network on which an *s*-*t* flow has been computed. The capacity of each edge appears as a label next to the edge, and the numbers in boxes give the amount of flow sent on each edge. (Edges without boxed numbers have no flow being sent on them.)

(a) What is the value of this flow? Is this a maximum (s,t) flow in this graph?

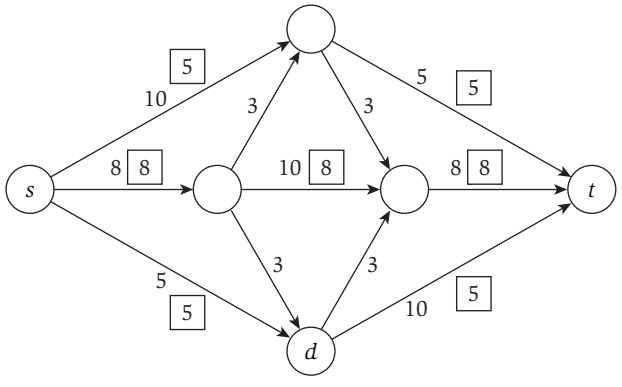

Figure 7.26 What is the value of the depicted flow? Is it a maximum flow? What is the minimum cut?

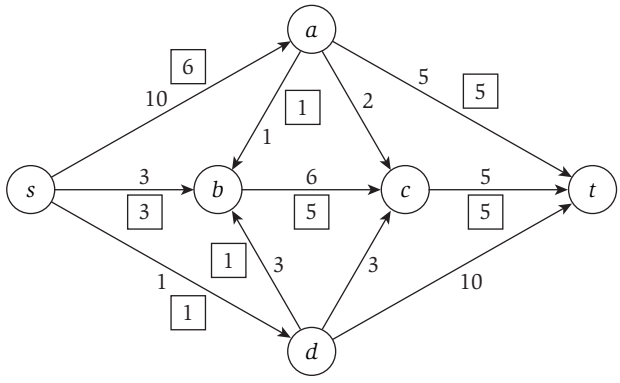

Figure 7.27 What is the value of the depicted flow? Is it a maximum flow? What is the minimum cut?

(b) Find a minimum *s*-*t* cut in the flow network pictured in Figure 7.27, and also say what its capacity is.

4. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary flow network, with a source s, a sink t, and a positive integer capacity c_e on every edge e. If f is a maximum s-t flow in G, then f saturates every edge out of s with flow (i.e., for all edges e out of s, we have $f(e) = c_e$).

5. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary flow network, with a source s, a sink t, and a positive integer capacity c_e on every edge e; and let (A, B) be a mimimum s-t cut with respect to these capacities $\{c_e : e \in E\}$. Now suppose we add 1 to every capacity; then (A, B) is still a minimum s-t cut with respect to these new capacities $\{1 + c_e : e \in E\}$.

6. Suppose you're a consultant for the Ergonomic Architecture Commission, and they come to you with the following problem.

They're really concerned about designing houses that are "user-friendly," and they've been having a lot of trouble with the setup of light fixtures and switches in newly designed houses. Consider, for example, a one-floor house with n light fixtures and n locations for light switches mounted in the wall. You'd like to be able to wire up one switch to control each light fixture, in such a way that a person at the switch can *see* the light fixture being controlled.

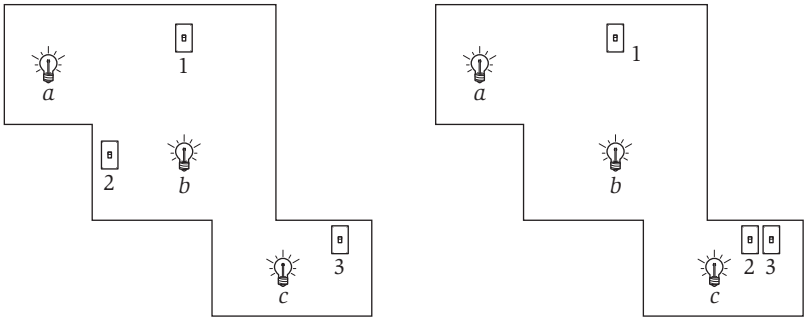

Figure 7.28 The floor plan in (a) is ergonomic, because we can wire switches to fixtures in such a way that each fixture is visible from the switch that controls it. (This can be done by wiring switch 1 to a, switch 2 to b, and switch 3 to c.) The floor plan in (b) is not ergonomic, because no such wiring is possible.

Sometimes this is possible and sometimes it isn't. Consider the two simple floor plans for houses in Figure 7.28. There are three light fixtures (labeled a, b, c) and three switches (labeled 1, 2, 3). It is possible to wire switches to fixtures in Figure 7.28(a) so that every switch has a line of sight to the fixture, but this is not possible in Figure 7.28(b).

Let's call a floor plan, together with n light fixture locations and n switch locations, *ergonomic* if it's possible to wire one switch to each fixture so that every fixture is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments in the plane (the walls), where the i^{th} wall has endpoints (x_i, y_i), (x'_i, y'_i). Each of the n switches and each of the n fixtures is given by its coordinates in the plane. A fixture is *visible* from a switch if the line segment joining them does not cross any of the walls.

Give an algorithm to decide if a given floor plan is ergonomic. The running time should be polynomial in m and n. You may assume that you have a subroutine with $O(1)$ running time that takes two line segments as input and decides whether or not they cross in the plane.

7. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible *base stations*. We'll suppose there are n clients, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well.

For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways.

There is a *range parameter* r—a client can only be connected to a base station that is within distance r. There is also a *load parameter* L—no more than L clients can be connected to any single base station.

Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

8. Statistically, the arrival of spring typically results in increased accidents and increased need for emergency medical treatment, which often requires blood transfusions. Consider the problem faced by a hospital that is trying to evaluate whether its blood supply is sufficient.

The basic rule for blood donation is the following. A person's own blood supply has certain *antigens* present (we can think of antigens as a kind of molecular signature); and a person cannot receive blood with a particular antigen if their own blood does not have this antigen present. Concretely, this principle underpins the division of blood into four *types*: A, B, AB, and O. Blood of type A has the A antigen, blood of type B has the B antigen, blood of type AB has both, and blood of type O has neither. Thus, patients with type A can receive only blood types A or O in a transfusion, patients with type B can receive only B or O, patients with type O can receive only O, and patients with type AB can receive any of the four types.[4]

(a) Let s_O, s_A, s_B, and s_{AB} denote the supply in whole units of the different blood types on hand. Assume that the hospital knows the projected demand for each blood type d_O, d_A, d_B, and d_{AB} for the coming week. Give a polynomial-time algorithm to evaluate if the blood on hand would suffice for the projected need.

(b) Consider the following example. Over the next week, they expect to need at most 100 units of blood. The typical distribution of blood types in U.S. patients is roughly 45 percent type O, 42 percent type A, 10 percent type B, and 3 percent type AB. The hospital wants to know if the blood supply it has on hand would be enough if 100 patients arrive with the expected type distribution. There is a total of 105 units of blood on hand. The table below gives these demands, and the supply on hand.

[4] The Austrian scientist Karl Landsteiner received the Nobel Prize in 1930 for his discovery of the blood types A, B, O, and AB.

blood type	supply	demand
O	50	45
A	36	42
B	11	8
AB	8	3

Is the 105 units of blood on hand enough to satisfy the 100 units of demand? Find an allocation that satisfies the maximum possible number of patients. Use an argument based on a minimum-capacity cut to show why not all patients can receive blood. Also, provide an explanation for this fact that would be understandable to the clinic administrators, who have not taken a course on algorithms. (So, for example, this explanation should not involve the words *flow*, *cut*, or *graph* in the sense we use them in this book.)

9. Network flow issues come up in dealing with natural disasters and other crises, since major unexpected events often require the movement and evacuation of large numbers of people in a short amount of time.

Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location (so different people will have different options for hospitals, depending on where they are right now).

At the same time, one doesn't want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is *balanced*: Each hospital receives at most $\lceil n/k \rceil$ people.

Give a polynomial-time algorithm that takes the given information about the people's locations and determines whether this is possible.

10. Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e, a source $s \in V$, and a sink $t \in V$. You are also given a maximum s-t flow in G, defined by a flow value f_e on each edge e. The flow f is *acyclic:* There is no cycle in G on which all edges carry positive flow. The flow f is also integer-valued.

Now suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m+n)$, where m is the number of edges in G and n is the number of nodes.

11. Your friends have written a very fast piece of maximum-flow code based on repeatedly finding augmenting paths as in Section 7.1. However, after you've looked at a bit of output from it, you realize that it's not always finding a flow of *maximum* value. The bug turns out to be pretty easy to find; your friends hadn't really gotten into the whole backward-edge thing when writing the code, and so their implementation builds a variant of the residual graph that *only includes the forward edges*. In other words, it searches for s-t paths in a graph $\tilde{G}_f$ consisting only of edges e for which $f(e) < c_e$, and it terminates when there is no augmenting path consisting entirely of such edges. We'll call this the Forward-Edge-Only Algorithm. (Note that we do not try to prescribe how this algorithm chooses its forward-edge paths; it may choose them in any fashion it wants, provided that it terminates only when there are no forward-edge paths.)

It's hard to convince your friends they need to reimplement the code. In addition to its blazing speed, they claim, in fact, that it never returns a flow whose value is less than a fixed fraction of optimal. Do you believe this? The crux of their claim can be made precise in the following statement.

> *There is an absolute constant $b > 1$ (independent of the particular input flow network), so that on every instance of the Maximum-Flow Problem, the Forward-Edge-Only Algorithm is guaranteed to find a flow of value at least $1/b$ times the maximum-flow value (regardless of how it chooses its forward-edge paths).*

Decide whether you think this statement is true or false, and give a proof of either the statement or its negation.

12. Consider the following problem. You are given a flow network with unit-capacity edges: It consists of a directed graph $G = (V, E)$, a source $s \in V$, and a sink $t \in V$; and $c_e = 1$ for every $e \in E$. You are also given a parameter k.

The goal is to delete k edges so as to reduce the maximum s-t flow in G by as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum s-t flow in $G' = (V, E - F)$ is as small as possible subject to this.

Give a polynomial-time algorithm to solve this problem.

13. In a standard s-t Maximum-Flow Problem, we assume edges have capacities, and there is no limit on how much flow is allowed to pass through a

node. In this problem, we consider the variant of the Maximum-Flow and Minimum-Cut problems with node capacities.

Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and nonnegative node capacities $\{c_v \geq 0\}$ for each $v \in V$. Given a flow f in this graph, the flow though a node v is defined as $f^{\text{in}}(v)$. We say that a flow is feasible if it satisfies the usual flow-conservation constraints and the node-capacity constraints: $f^{\text{in}}(v) \leq c_v$ for all nodes.

Give a polynomial-time algorithm to find an s-t maximum flow in such a node-capacitated network. Define an s-t cut for node-capacitated networks, and show that the analogue of the Max-Flow Min-Cut Theorem holds true.

14. We define the *Escape Problem* as follows. We are given a directed graph $G = (V, E)$ (picture a network of roads). A certain collection of nodes $X \subset V$ are designated as *populated nodes*, and a certain other collection $S \subset V$ are designated as *safe nodes*. (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that (i) each node in X is the tail of one path, (ii) the last node on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.

(a) Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.

(b) Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus we change (iii) to say "the paths do not share any *nodes*."

With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists.

Also, provide an example with the same G, X, and S, in which the answer is yes to the question in (a) but no to the question in (b).

15. Suppose you and your friend Alanis live, together with $n - 2$ other people, at a popular off-campus cooperative apartment, the Upson Collective. Over the next n nights, each of you is supposed to cook dinner for the co-op exactly once, so that someone cooks on each of the nights.

Of course, everyone has scheduling conflicts with some of the nights (e.g., exams, concerts, etc.), so deciding who should cook on which night becomes a tricky task. For concreteness, let's label the people

$$\{p_1, \ldots, p_n\},$$

the nights

$$\{d_1, \ldots, d_n\};$$

and for person p_i, there's a set of nights $S_i \subset \{d_1, \ldots, d_n\}$ when they are *not* able to cook.

A *feasible dinner schedule* is an assignment of each person in the co-op to a different night, so that each person cooks on exactly one night, there is someone cooking on each night, and if p_i cooks on night d_j, then $d_j \notin S_i$.

(a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible dinner schedule for the co-op.

(b) Your friend Alanis takes on the task of trying to construct a feasible dinner schedule. After great effort, she constructs what she claims is a feasible schedule and then heads off to class for the day.

Unfortunately, when you look at the schedule she created, you notice a big problem. $n - 2$ of the people at the co-op are assigned to different nights on which they are available: no problem there. But for the other two people, p_i and p_j, and the other two days, d_k and d_ℓ, you discover that she has accidentally assigned both p_i and p_j to cook on night d_k, and assigned no one to cook on night d_ℓ.

You want to fix Alanis's mistake but without having to recompute everything from scratch. Show that it's possible, using her "almost correct" schedule, to decide in only $O(n^2)$ time whether there exists a feasible dinner schedule for the co-op. (If one exists, you should also output it.)

16. Back in the euphoric early days of the Web, people liked to claim that much of the enormous potential in a company like Yahoo! was in the "eyeballs"—the simple fact that millions of people look at its pages every day. Further, by convincing people to register personal data with the site, a site like Yahoo! can show each user an extremely targeted advertisement whenever he or she visits the site, in a way that TV networks or magazines couldn't hope to match. So if a user has told Yahoo! that he or she is a 20-year-old computer science major from Cornell University, the site can present a banner ad for apartments in Ithaca, New York; on the other hand, if he or she is a 50-year-old investment banker from Greenwich, Connecticut, the site can display a banner ad pitching Lincoln Town Cars instead.

But deciding on which ads to show to which people involves some serious computation behind the scenes. Suppose that the managers of a popular Web site have identified k distinct *demographic groups*

$G_1, G_2, \ldots, G_k$. (These groups can overlap; for example, G_1 can be equal to all residents of New York State, and G_2 can be equal to all people with a degree in computer science.) The site has contracts with m different *advertisers*, to show a certain number of copies of their ads to users of the site. Here's what the contract with the i^{th} advertiser looks like.

- For a subset $X_i \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i.
- For a number r_i, advertiser i wants its ads shown to at least r_i users each minute.

Now consider the problem of designing a good *advertising policy*—a way to show a single ad to each user of the site. Suppose at a given minute, there are n users visiting the site. Because we have registration information on each of these users, we know that user j (for $j = 1, 2, \ldots, n$) belongs to a subset $U_j \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups. The problem is: Is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute? (That is, for each $i = 1, 2, \ldots, m$, can at least r_i of the n users, each belonging to at least one demographic group in X_i, be shown an ad provided by advertiser i?)

Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

17. You've been called in to help some network administrators diagnose the extent of a failure in their network. The network is designed to carry traffic from a designated source node s to a designated target node t, so we will model the network as a directed graph $G = (V, E)$, in which the capacity of each edge is 1 and in which each node lies on at least one path from s to t.

Now, when everything is running smoothly in the network, the maximum s-t flow in G has value k. However, the current situation (and the reason you're here) is that an attacker has destroyed some of the edges in the network, so that there is now no path from s to t using the remaining (surviving) edges. For reasons that we won't go into here, they believe the attacker has destroyed only k edges, the minimum number needed to separate s from t (i.e., the size of a minimum s-t cut); and we'll assume they're correct in believing this.

The network administrators are running a monitoring tool on node s, which has the following behavior. If you issue the command $ping(v)$, for a given node v, it will tell you whether there is currently a path from s to v. (So $ping(t)$ reports that no path currently exists; on the other hand,

ping(s) always reports a path from s to itself.) Since it's not practical to go out and inspect every edge of the network, they'd like to determine the extent of the failure using this monitoring tool, through judicious use of the *ping* command.

So here's the problem you face: Give an algorithm that issues a sequence of *ping* commands to various nodes in the network and then reports the *full* set of nodes that are not currently reachable from s. You could do this by pinging every node in the network, of course, but you'd like to do it using many fewer pings (given the assumption that only k edges have been deleted). In issuing this sequence, your algorithm is allowed to decide which node to ping next based on the outcome of earlier *ping* operations.

Give an algorithm that accomplishes this task using only $O(k \log n)$ pings.

18. We consider the Bipartite Matching Problem on a bipartite graph $G = (V, E)$. As usual, we say that V is partitioned into sets X and Y, and each edge has one end in X and the other in Y.

If M is a matching in G, we say that a node $y \in Y$ is *covered* by M if y is an end of one of the edges in M.

(a) Consider the following problem. We are given G and a matching M in G. For a given number k, we want to decide if there is a matching M' in G so that

(i) M' has k more edges than M does, *and*

(ii) every node $y \in Y$ that is covered by M is also covered by M'.

We call this the *Coverage Expansion Problem*, with input G, M, and k. and we will say that M' is a *solution* to the instance.

Give a polynomial-time algorithm that takes an instance of Coverage Expansion and either returns a solution M' or reports (correctly) that there is no solution. (You should include an analysis of the running time and a brief proof of why it is correct.)

Note: You may wish to also look at part (b) to help in thinking about this.

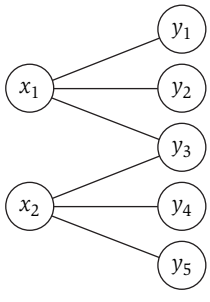

Figure 7.29 An instance of Coverage Expansion.

Example. Consider Figure 7.29, and suppose M is the matching consisting of the edge (x_1, y_2). Suppose we are asked the above question with $k = 1$.

Then the answer to this instance of Coverage Expansion is yes. We can let M' be the matching consisting (for example) of the two edges (x_1, y_2) and (x_2, y_4); M' has one more edge than M, and y_2 is still covered by M'.

(b) Give an example of an instance of Coverage Expansion, specified by G, M, and k, so that the following situation happens.

The instance has a solution; but in any solution M', the edges of M do not form a subset of the edges of M'.

(c) Let G be a bipartite graph, and let M be any matching in G. Consider the following two quantities.

- K_1 is the size of the largest matching M' so that every node y that is covered by M is also covered by M'.
- K_2 is the size of the largest matching M'' in G.

Clearly $K_1 \leq K_2$, since K_2 is obtained by considering *all possible* matchings in G.

Prove that in fact $K_1 = K_2$; that is, we can obtain a maximum matching even if we're constrained to cover all the nodes covered by our initial matching M.

19. You've periodically helped the medical consulting firm Doctors Without Weekends on various hospital scheduling issues, and they've just come to you with a new problem. For each of the next n days, the hospital has determined the number of doctors they want on hand; thus, on day i, they have a requirement that *exactly* p_i doctors be present.

There are k doctors, and each is asked to provide a list of days on which he or she is willing to work. Thus doctor j provides a set L_j of days on which he or she is willing to work.

The system produced by the consulting firm should take these lists and try to return to each doctor j a list L'_j with the following properties.

(A) L'_j is a subset of L_j, so that doctor j only works on days he or she finds acceptable.

(B) If we consider the whole set of lists $L'_1, \ldots, L'_k$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

(a) Describe a polynomial-time algorithm that implements this system. Specifically, give a polynomial-time algorithm that takes the numbers $p_1, p_2, \ldots, p_n$, and the lists $L_1, \ldots, L_k$, and does one of the following two things.

- Return lists $L'_1, L'_2, \ldots, L'_k$ satisfying properties (A) and (B); or
- Report (correctly) that there is no set of lists $L'_1, L'_2, \ldots, L'_k$ that satisfies both properties (A) and (B).

(b) The hospital finds that the doctors tend to submit lists that are much too restrictive, and so it often happens that the system reports (correctly, but unfortunately) that no acceptable set of lists $L'_1, L'_2, \ldots, L'_k$ exists.

Thus the hospital relaxes the requirements as follows. They add a new parameter $c > 0$, and the system now should try to return to each doctor j a list L'_j with the following properties.

(A*) L'_j contains at most c days that do not appear on the list L_j.

(B) *(Same as before)* If we consider the whole set of lists $L'_1, \ldots, L'_k$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

Describe a polynomial-time algorithm that implements this revised system. It should take the numbers $p_1, p_2, \ldots, p_n$, the lists $L_1, \ldots, L_k$, and the parameter $c > 0$, and do one of the following two things.

- Return lists $L'_1, L'_2, \ldots, L'_k$ satisfying properties (A*) and (B); or
- Report (correctly) that there is no set of lists $L'_1, L'_2, \ldots, L'_k$ that satisfies both properties (A*) and (B).

20. Your friends are involved in a large-scale atmospheric science experiment. They need to get good measurements on a set S of n different conditions in the atmosphere (such as the ozone level at various places), and they have a set of m balloons that they plan to send up to make these measurements. Each balloon can make at most two measurements.

Unfortunately, not all balloons are capable of measuring all conditions, so for each balloon $i = 1, \ldots, m$, they have a set S_i of conditions that balloon i can measure. Finally, to make the results more reliable, they plan to take each measurement from at least k different balloons. (Note that a single balloon should not measure the same condition twice.) They are having trouble figuring out which conditions to measure on which balloon.

Example. Suppose that $k = 2$, there are $n = 4$ conditions labeled c_1, c_2, c_3, c_4, and there are $m = 4$ balloons that can measure conditions, subject to the limitation that $S_1 = S_2 = \{c_1, c_2, c_3\}$, and $S_3 = S_4 = \{c_1, c_3, c_4\}$. Then one possible way to make sure that each condition is measured at least $k = 2$ times is to have

- balloon 1 measure conditions c_1, c_2,
- balloon 2 measure conditions c_2, c_3,
- balloon 3 measure conditions c_3, c_4, and
- balloon 4 measure conditions c_1, c_4.

(a) Give a polynomial-time algorithm that takes the input to an instance of this problem (the n conditions, the sets S_i for each of the m balloons, and the parameter k) and decides whether there is a way to measure each condition by k different balloons, while each balloon only measures at most two conditions.

(b) You show your friends a solution computed by your algorithm from (a), and to your surprise they reply, "This won't do at all—one of the conditions is only being measured by balloons from a single subcontractor." You hadn't heard anything about subcontractors before; it turns out there's an extra wrinkle they forgot to mention. . . .

Each of the balloons is produced by one of three different subcontractors involved in the experiment. A requirement of the experiment is that there be no condition for which all k measurements come from balloons produced by a single subcontractor.

For example, suppose balloon 1 comes from the first subcontractor, balloons 2 and 3 come from the second subcontractor, and balloon 4 comes from the third subcontractor. Then our previous solution no longer works, as both of the measurements for condition c_3 were done by balloons from the second subcontractor. However, we could use balloons 1 and 2 to each measure conditions c_1, c_2, and use balloons 3 and 4 to each measure conditions c_3, c_4.

Explain how to modify your polynomial-time algorithm for part (a) into a new algorithm that decides whether there exists a solution satisfying all the conditions from (a), plus the new requirement about subcontractors.

21. You're helping to organize a class on campus that has decided to give all its students wireless laptops for the semester. Thus there is a collection of n wireless laptops; there is also have a collection of n wireless *access points*, to which a laptop can connect when it is in range.

The laptops are currently scattered across campus; laptop ℓ is within range of a *set* S_ℓ of access points. We will assume that each laptop is within range of at least one access point (so the sets S_ℓ are nonempty); we will also assume that every access point p has at least one laptop within range of it.

To make sure that all the wireless connectivity software is working correctly, you need to try having laptops make contact with access points in such a way that each laptop and each access point is involved in at least one connection. Thus we will say that a *test set* T is a collection of ordered pairs of the form (ℓ, p), for a laptop ℓ and access point p, with the properties that

(i) If $(\ell, p) \in T$, then ℓ is within range of p (i.e., $p \in S_\ell$).

(ii) Each laptop appears in at least one ordered pair in T.

(iii) Each access point appears in at least one ordered pair in T.

This way, by trying out all the connections specified by the pairs in T, we can be sure that each laptop and each access point have correctly functioning software.

The problem is: Given the sets S_ℓ for each laptop (i.e., which laptops are within range of which access points), and a number k, decide whether there is a test set of size at most k.

Example. Suppose that $n = 3$; laptop 1 is within range of access points 1 and 2; laptop 2 is within range of access point 2; and laptop 3 is within range of access points 2 and 3. Then the set of pairs

(laptop 1, access point 1), (laptop 2, access point 2),
(laptop 3, access point 3)

would form a test set of size 3.

(a) Give an example of an instance of this problem for which there is no test set of size n. (Recall that we assume each laptop is within range of at least one access point, and each access point p has at least one laptop within range of it.)

(b) Give a polynomial-time algorithm that takes the input to an instance of this problem (including the parameter k) and decides whether there is a test set of size at most k.

22. Let M be an $n \times n$ matrix with each entry equal to either 0 or 1. Let m_{ij} denote the entry in row i and column j. A *diagonal entry* is one of the form m_{ii} for some i.

Swapping rows i and j of the matrix M denotes the following action: we swap the values m_{ik} and m_{jk} for $k = 1, 2, \ldots, n$. Swapping two columns is defined analogously.

We say that M is *rearrangeable* if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that, after all the swapping, all the diagonal entries of M are equal to 1.

(a) Give an example of a matrix M that is not rearrangeable, but for which at least one entry in each row and each column is equal to 1.

(b) Give a polynomial-time algorithm that determines whether a matrix M with 0-1 entries is rearrangeable.

23. Suppose you're looking at a flow network G with source s and sink t, and you want to be able to express something like the following intuitive notion: Some nodes are clearly on the "source side" of the main bottlenecks; some nodes are clearly on the "sink side" of the main bottlenecks; and some nodes are in the middle. However, G can have many minimum cuts, so we have to be careful in how we try making this idea precise.

Here's one way to divide the nodes of G into three categories of this sort.

- We say a node v is *upstream* if, for all minimum s-t cuts (A, B), we have $v \in A$—that is, v lies on the source side of every minimum cut.
- We say a node v is *downstream* if, for all minimum s-t cuts (A, B), we have $v \in B$—that is, v lies on the sink side of every minimum cut.
- We say a node v is *central* if it is neither upstream nor downstream; there is at least one minimum s-t cut (A, B) for which $v \in A$, and at least one minimum s-t cut (A', B') for which $v \in B'$.

Give an algorithm that takes a flow network G and classifies each of its nodes as being upstream, downstream, or central. The running time of your algorithm should be within a constant factor of the time required to compute a *single* maximum flow.

24. Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and nonnegative edge capacities $\{c_e\}$. Give a polynomial-time algorithm to decide whether G has a *unique* minimum s-t cut (i.e., an s-t of capacity strictly less than that of all other s-t cuts).

25. Suppose you live in a big apartment with a lot of friends. Over the course of a year, there are many occasions when one of you pays for an expense shared by some subset of the apartment, with the expectation that everything will get balanced out fairly at the end of the year. For example, one of you may pay the whole phone bill in a given month, another will occasionally make communal grocery runs to the nearby organic food emporium, and a third might sometimes use a credit card to cover the whole bill at the local Italian-Indian restaurant, Little Idli.

In any case, it's now the end of the year and time to settle up. There are n people in the apartment; and for each ordered pair (i, j) there's an amount $a_{ij} \geq 0$ that i owes j, accumulated over the course of the year. We will require that for any two people i and j, at least one of the quantities a_{ij} or a_{ji} is equal to 0. This can be easily made to happen as follows: If it turns out that i owes j a positive amount x, and j owes i a positive amount $y < x$, then we will subtract off y from both sides and declare $a_{ij} = x - y$ while $a_{ji} = 0$. In terms of all these quantities, we now define the *imbalance* of a person i to be the sum of the amounts that i is owed by everyone else, minus the sum of the amounts that i owes everyone else. (Note that an imbalance can be positive, negative, or zero.)

In order to restore all imbalances to 0, so that everyone departs on good terms, certain people will write checks to others; in other words, for certain ordered pairs (i, j), i will write a check to j for an amount $b_{ij} > 0$.

We will say that a set of checks constitutes a *reconciliation* if, for each person i, the total value of the checks received by i, minus the total value of the checks written by i, is equal to the imbalance of i. Finally, you and your friends feel it is bad form for i to write j a check if i did not actually owe j money, so we say that a reconciliation is *consistent* if, whenever i writes a check to j, it is the case that $a_{ij} > 0$.

Show that, for any set of amounts a_{ij}, there is always a consistent reconciliation in which at most $n - 1$ checks get written, by giving a polynomial-time algorithm to compute such a reconciliation.

26. You can tell that cellular phones are at work in rural communities, from the giant microwave towers you sometimes see sprouting out of corn fields and cow pastures. Let's consider a very simplified model of a cellular phone network in a sparsely populated area.

We are given the locations of n *base stations*, specified as points $b_1, \ldots, b_n$ in the plane. We are also given the locations of n cellular phones, specified as points $p_1, \ldots, p_n$ in the plane. Finally, we are given a *range parameter* $\Delta > 0$. We call the set of cell phones *fully connected* if it is possible to assign each phone to a base station in such a way that

- Each phone is assigned to a different base station, and
- If a phone at p_i is assigned to a base station at b_j, then the straight-line distance between the points p_i and b_j is at most Δ.

Suppose that the owner of the cell phone at point p_1 decides to go for a drive, traveling continuously for a total of z units of distance due east. As this cell phone moves, we may have to update the assignment of phones to base stations (possibly several times) in order to keep the set of phones fully connected.

Give a polynomial-time algorithm to decide whether it is possible to keep the set of phones fully connected at all times during the travel of this one cell phone. (You should assume that all other phones remain stationary during this travel.) If it is possible, you should report a sequence of assignments of phones to base stations that will be sufficient in order to maintain full connectivity; if it is not possible, you should report a point on the traveling phone's path at which full connectivity cannot be maintained.

You should try to make your algorithm run in $O(n^3)$ time if possible.

Example. Suppose we have phones at $p_1 = (0, 0)$ and $p_2 = (2, 1)$; we have base stations at $b_1 = (1, 1)$ and $b_2 = (3, 1)$; and $\Delta = 2$. Now consider the case in which the phone at p_1 moves due east a distance of 4 units, ending at $(4, 0)$. Then it is possible to keep the phones fully connected during this

motion: We begin by assigning p_1 to b_1 and p_2 to b_2, and we reassign p_1 to b_2 and p_2 to b_1 during the motion (for example, when p_1 passes the point $(2, 0)$).

27. Some of your friends with jobs out West decide they really need some extra time each day to sit in front of their laptops, and the morning commute from Woodside to Palo Alto seems like the only option. So they decide to carpool to work.

Unfortunately, they all hate to drive, so they want to make sure that any carpool arrangement they agree upon is fair and doesn't overload any individual with too much driving. Some sort of simple round-robin scheme is out, because none of them goes to work every day, and so the subset of them in the car varies from day to day.

Here's one way to define *fairness*. Let the people be labeled $S = \{p_1, \ldots, p_k\}$. We say that the *total driving obligation* of p_j over a set of days is the expected number of times that p_j would have driven, had a driver been chosen uniformly at random from among the people going to work each day. More concretely, suppose the carpool plan lasts for d days, and on the i^{th} day a subset $S_i \subseteq S$ of the people go to work. Then the above definition of the total driving obligation Δ_j for p_j can be written as $\Delta_j = \sum_{i:p_j \in S_i} \frac{1}{|S_i|}$. Ideally, we'd like to require that p_j drives at most Δ_j times; unfortunately, Δ_j may not be an integer.

So let's say that a *driving schedule* is a choice of a driver for each day—that is, a sequence $p_{i_1}, p_{i_2}, \ldots, p_{i_d}$ with $p_{i_t} \in S_t$—and that a *fair driving schedule* is one in which each p_j is chosen as the driver on at most $\lceil \Delta_j \rceil$ days.

(a) Prove that for any sequence of sets $S_1, \ldots, S_d$, there exists a fair driving schedule.

(b) Give an algorithm to compute a fair driving schedule with running time polynomial in k and d.

28. A group of students has decided to add some features to Cornell's on-line Course Management System (CMS), to handle aspects of course planning that are not currently covered by the software. They're beginning with a module that helps schedule office hours at the start of the semester.

Their initial prototype works as follows. The office hour schedule will be the same from one week to the next, so it's enough to focus on the scheduling problem for a single week. The course administrator enters a collection of nonoverlapping one-hour time intervals $I_1, I_2, \ldots, I_k$ when it would be possible for teaching assistants (TAs) to hold office hours; the eventual office-hour schedule will consist of a subset of some, but

generally not all, of these time slots. Then each of the TAs enters his or her weekly schedule, showing the times when he or she would be available to hold office hours.

Finally, the course administrator specifies, for parameters a, b, and c, that they would like each TA to hold between a and b office hours per week, and they would like a total of exactly c office hours to be held over the course of the week.

The problem, then, is how to assign each TA to some of the office-hour time slots, so that each TA is available for each of his or her office-hour slots, and so that the right number of office hours gets held. (There should be only one TA at each office hour.)

Example. Suppose there are five possible time slots for office hours:

I_1 = *Mon 3–4 P.M.*; I_2 = *Tue 1–2 P.M.*; I_3 = *Wed 10–11 A.M.*; I_4 = *Wed 3–4 P.M.*; *and* I_5 = *Thu 10–11 A.M.*.

There are two TAs; the first would be able to hold office hours at any time on Monday or Wednesday afternoons, and the second would be able to hold office hours at any time on Tuesday, Wednesday, or Thursday. (In general, TA availability might be more complicated to specify than this, but we're keeping this example simple.) Finally, each TA should hold between $a = 1$ and $b = 2$ office hours, and we want exactly $c = 3$ office hours per week total.

One possible solution would be to have the first TA hold office hours in time slot I_1, and the second TA to hold office hours in time slots I_2 and I_5.

(a) Give a polynomial-time algorithm that takes the input to an instance of this problem (the time slots, the TA schedules, and the parameters a, b, and c) and does one of the following two things:
- Constructs a valid schedule for office hours, specifying which TA will cover which time slots, or
- Reports (correctly) that there is no valid way to schedule office hours.

(b) This office-hour scheduling feature becomes very popular, and so course staffs begin to demand more. In particular, they observe that it's good to have a greater density of office hours closer to the due date of a homework assignment.

So what they want to be able to do is to specify an *office-hour density* parameter for each day of the week: The number d_i specifies that they want to have at least d_i office hours on a given day i of the week.

For example, suppose that in our previous example, we add the constraint that we want at least one office hour on Wednesday and at least one office hour on Thursday. Then the previous solution does not work; but there is a possible solution in which we have the first TA hold office hours in time slot I_1, and the second TA hold office hours in time slots I_3 and I_5. (Another solution would be to have the first TA hold office hours in time slots I_1 and I_4, and the second TA hold office hours in time slot I_5.)

Give a polynomial-time algorithm that computes office-hour schedules under this more complex set of constraints. The algorithm should either construct a schedule or report (correctly) that none exists.

29. Some of your friends have recently graduated and started a small company, which they are currently running out of their parents' garages in Santa Clara. They're in the process of porting all their software from an old system to a new, revved-up system; and they're facing the following problem.

They have a collection of n software applications, $\{1, 2, \ldots, n\}$, running on their old system; and they'd like to port some of these to the new system. If they move application i to the new system, they expect a net (monetary) benefit of $b_i \geq 0$. The different software applications interact with one another; if applications i and j have extensive interaction, then the company will incur an expense if they move one of i or j to the new system but not both; let's denote this expense by $x_{ij} \geq 0$.

So, if the situation were really this simple, your friends would just port all n applications, achieving a total benefit of $\sum_i b_i$. Unfortunately, there's a problem. . . .

Due to small but fundamental incompatibilities between the two systems, there's no way to port application 1 to the new system; it will have to remain on the old system. Nevertheless, it might still pay off to port some of the other applications, accruing the associated benefit and incurring the expense of the interaction between applications on different systems.

So this is the question they pose to you: Which of the remaining applications, if any, should be moved? Give a polynomial-time algorithm to find a set $S \subseteq \{2, 3, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

30. Consider a variation on the previous problem. In the new scenario, any application can potentially be moved, but now some of the benefits b_i for

moving to the new system are in fact *negative*: If $b_i < 0$, then it is preferable (by an amount quantified in b_i) to keep i on the old system. Again, give a polynomial-time algorithm to find a set $S \subseteq \{1, 2, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

31. Some of your friends are interning at the small high-tech company WebExodus. A running joke among the employees there is that the back room has less space devoted to high-end servers than it does to empty boxes of computer equipment, piled up in case something needs to be shipped back to the supplier for maintainence.

A few days ago, a large shipment of computer monitors arrived, each in its own large box; and since there are many different kinds of monitors in the shipment, the boxes do not all have the same dimensions. A bunch of people spent some time in the morning trying to figure out how to store all these things, realizing of course that less space would be taken up if some of the boxes could be *nested* inside others.

Suppose each box i is a rectangular parallelepiped with side lengths equal to (i_1, i_2, i_3); and suppose each side length is strictly between half a meter and one meter. Geometrically, you know what it means for one box to nest inside another: It's possible if you can rotate the smaller so that it fits inside the larger in each dimension. Formally, we can say that box i with dimensions (i_1, i_2, i_3) *nests* inside box j with dimensions (j_1, j_2, j_3) if there is a permutation a, b, c of the dimensions $\{1, 2, 3\}$ so that $i_a < j_1$, and $i_b < j_2$, and $i_c < j_3$. Of course, nesting is recursive: If i nests in j, and j nests in k, then by putting i inside j inside k, only box k is visible. We say that a *nesting arrangement* for a set of n boxes is a sequence of operations in which a box i is put inside another box j in which it nests; and if there were already boxes nested inside i, then these end up inside j as well. (Also notice the following: Since the side lengths of i are more than half a meter each, and since the side lengths of j are less than a meter each, box i will take up more than half of each dimension of j, and so after i is put inside j, nothing else can be put inside j.) We say that a box k is *visible* in a nesting arrangement if the sequence of operations does not result in its ever being put inside another box.

Here is the problem faced by the people at WebExodus: Since only the visible boxes are taking up any space, how should a nesting arrangement be chosen so as to minimize the *number* of visible boxes?

Give a polynomial-time algorithm to solve this problem.

Example. Suppose there are three boxes with dimensions $(.6, .6, .6)$, $(.75, .75, .75)$, and $(.9, .7, .7)$. The first box can be put into either of the

second or third boxes; but in any nesting arrangement, both the second and third boxes will be visible. So the minimum possible number of visible boxes is two, and one solution that achieves this is to nest the first box inside the second.

32. Given a graph $G = (V, E)$, and a natural number k, we can define a relation $\xrightarrow{G,k}$ on pairs of vertices of G as follows. If $x, y \in V$, we say that $x \xrightarrow{G,k} y$ if there exist k mutually edge-disjoint paths from x to y in G.

Is it true that for every G and every $k \geq 0$, the relation $\xrightarrow{G,k}$ is transitive? That is, is it always the case that if $x \xrightarrow{G,k} y$ and $y \xrightarrow{G,k} z$, then we have $x \xrightarrow{G,k} z$? Give a proof or a counterexample.

33. Let $G = (V, E)$ be a directed graph, and suppose that for each node v, the number of edges into v is equal to the number of edges out of v. That is, for all v,

$$|\{(u, v) : (u, v) \in E\}| = |\{(v, w) : (v, w) \in E\}|.$$

Let x, y be two nodes of G, and suppose that there exist k mutually edge-disjoint paths from x to y. Under these conditions, does it follow that there exist k mutually edge-disjoint paths from y to x? Give a proof or a counterexample with explanation.

34. *Ad hoc networks*, made up of low-powered wireless devices, have been proposed for situations like natural disasters in which the coordinators of a rescue effort might want to monitor conditions in a hard-to-reach area. The idea is that a large collection of these wireless devices could be dropped into such an area from an airplane and then configured into a functioning network.

Note that we're talking about (a) relatively inexpensive devices that are (b) being dropped from an airplane into (c) dangerous territory; and for the combination of reasons (a), (b), and (c), it becomes necessary to include provisions for dealing with the failure of a reasonable number of the nodes.

We'd like it to be the case that if one of the devices v detects that it is in danger of failing, it should transmit a representation of its current state to some other device in the network. Each device has a limited transmitting range—say it can communicate with other devices that lie within d meters of it. Moreover, since we don't want it to try transmitting its state to a device that has already failed, we should include some redundancy: A device v should have a set of k other devices that it can potentially contact, each within d meters of it. We'll call this a *back-up set* for device v.

(a) Suppose you're given a set of n wireless devices, with positions represented by an (x, y) coordinate pair for each. Design an algorithm that determines whether it is possible to choose a back-up set for each device (i.e., k other devices, each within d meters), with the further property that, for some parameter b, no device appears in the back-up set of more than b other devices. The algorithm should output the back-up sets themselves, provided they can be found.

(b) The idea that, for each pair of devices v and w, there's a strict dichotomy between being "in range" or "out of range" is a simplified abstraction. More accurately, there's a power decay function $f(\cdot)$ that specifies, for a pair of devices at distance δ, the signal strength $f(\delta)$ that they'll be able to achieve on their wireless connection. (We'll assume that $f(\delta)$ decreases with increasing δ.)

We might want to build this into our notion of back-up sets as follows: among the k devices in the back-up set of v, there should be at least one that can be reached with very high signal strength, at least one other that can be reached with moderately high signal strength, and so forth. More concretely, we have values $p_1 \geq p_2 \geq \cdots \geq p_k$, so that if the back-up set for v consists of devices at distances $d_1 \leq d_2 \leq \cdots \leq d_k$, then we should have $f(d_j) \geq p_j$ for each j.

Give an algorithm that determines whether it is possible to choose a back-up set for each device subject to this more detailed condition, still requiring that no device should appear in the back-up set of more than b other devices. Again, the algorithm should output the back-up sets themselves, provided they can be found.

35. You're designing an interactive image segmentation tool that works as follows. You start with the image segmentation setup described in Section 7.10, with n pixels, a set of neighboring pairs, and parameters $\{a_i\}$, $\{b_i\}$, and $\{p_{ij}\}$. We will make two assumptions about this instance. First, we will suppose that each of the parameters $\{a_i\}$, $\{b_i\}$, and $\{p_{ij}\}$ is a nonnegative integer between 0 and d, for some number d. Second, we will suppose that the neighbor relation among the pixels has the property that each pixel is a neighbor of at most four other pixels (so in the resulting graph, there are at most four edges out of each node).

You first perform an *initial segmentation* (A_0, B_0) so as to maximize the quantity $q(A_0, B_0)$. Now, this might result in certain pixels being assigned to the background when the user knows that they ought to be in the foreground. So, when presented with the segmentation, the user has the option of mouse-clicking on a particular pixel v_1, thereby bringing it to the foreground. But the tool should not simply bring this pixel into

the foreground; rather, it should compute a segmentation (A_1, B_1) that maximizes the quantity $q(A_1, B_1)$ *subject to the condition that* v_1 *is in the foreground.* (In practice, this is useful for the following kind of operation: In segmenting a photo of a group of people, perhaps someone is holding a bag that has been accidentally labeled as part of the background. By clicking on a single pixel belonging to the bag, and recomputing an optimal segmentation subject to the new condition, the whole bag will often become part of the foreground.)

In fact, the system should allow the user to perform a sequence of such mouse-clicks $v_1, v_2, \ldots, v_t$; and after mouse-click v_i, the system should produce a segmentation (A_i, B_i) that maximizes the quantity $q(A_i, B_i)$ subject to the condition that all of $v_1, v_2, \ldots, v_i$ are in the foreground.

Give an algorithm that performs these operations so that the initial segmentation is computed within a constant factor of the time for a single maximum flow, and then the interaction with the user is handled in $O(dn)$ time per mouse-click.

(*Note:* Solved Exercise 1 from this chapter is a useful primitive for doing this. Also, the symmetric operation of forcing a pixel to belong to the background can be handled by analogous means, but you do not have to work this out here.)

36. We now consider a different variation of the image segmentation problem in Section 7.10. We will develop a solution to an *image labeling* problem, where the goal is to label each pixel with a rough estimate of its distance from the camera (rather than the simple *foreground/background* labeling used in the text). The possible labels for each pixel will be $0, 1, 2, \ldots, M$ for some integer M.

Let $G = (V, E)$ denote the graph whose nodes are pixels, and edges indicate neighboring pairs of pixels. A *labeling* of the pixels is a partition of V into sets $A_0, A_1, \ldots, A_M$, where A_k is the set of pixels that is labeled with distance k for $k = 0, \ldots, M$. We will seek a labeling of minimum *cost*; the cost will come from two types of terms. By analogy with the foreground/background segmentation problem, we will have an *assignment cost*: for each pixel i and label k, the cost $a_{i,k}$ is the cost of assigning label k to pixel i. Next, if two neighboring pixels $(i, j) \in E$ are assigned different labels, there will be a *separation* cost. In Section 7.10, we used a separation penalty p_{ij}. In our current problem, the separation cost will also depend on how far the two pixels are separated; specifically, it will be proportional to the difference in value between their two labels.

Thus the overall cost q' of a labeling is defined as follows:

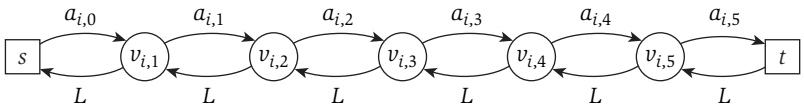

Figure 7.30 The set of nodes corresponding to a single pixel i in Exercise 36 (shown together with the source s and sink t).

$$q'(A_0, \ldots, A_M) = \sum_{k=0}^{M} \sum_{i \in A_i} a_{i,k} + \sum_{k<\ell} \sum_{\substack{(i,j)\in E \\ i\in A_k, j\in A_\ell}} (\ell - k)p_{ij}.$$

The goal of this problem is to develop a polynomial-time algorithm that finds the optimal labeling given the graph G and the penalty parameters $a_{i,k}$ and p_{ij}. The algorithm will be based on constructing a flow network, and we will start you off on designing the algorithm by providing a portion of the construction.

The flow network will have a source s and a sink t. In addition, for each pixel $i \in V$ we will have nodes $v_{i,k}$ in the flow network for $k = 1, \ldots, M$, as shown in Figure 7.30. ($M = 5$ in the example in the figure.)

For notational convenience, the nodes $v_{i,0}$ and $v_{i,M+1}$ will refer to s and t, respectively, for any choice of $i \in V$.

We now add edges $(v_{i,k}, v_{i,k+1})$ with capacity $a_{i,k}$ for $k = 0, \ldots, M$; and edges $(v_{i,k+1}, v_{i,k})$ in the opposite direction with very large capacity L. We will refer to this collection of nodes and edges as the *chain* associated with pixel i.

Notice that if we make this very large capacity L large enough, then there will be no minimum cut (A, B) so that an edge of capacity L leaves the set A. (How large do we have to make it for this to happen?). Hence, for any minimum cut (A, B), and each pixel i, there will be exactly one low-capacity edge in the chain associated with i that leaves the set A. (You should check that if there were two such edges, then a large-capacity edge would also have to leave the set A.)

Finally, here's the question: Use the nodes and edges defined so far to complete the construction of a flow network with the property that a minimum-cost labeling can be efficiently computed from a minimum s-t cut. You should prove that your construction has the desired property, and show how to recover the minimum-cost labeling from the cut.

37. In a standard minimum s-t cut problem, we assume that all capacities are nonnegative; allowing an arbitrary set of positive and negative capacities results in a problem that is computationally much more difficult. How-

ever, as we'll see here, it is possible to relax the nonnegativity requirement a little and still have a problem that can be solved in polynomial time.

Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and edge capacities $\{c_e\}$. Suppose that for every edge e that has neither s nor t as an endpoint, we have $c_e \geq 0$. Thus c_e can be negative for edges e that have at least one end equal to either s or t. Give a polynomial-time algorithm to find an s-t cut of minimum value in such a graph. (Despite the new nonnegativity requirements, we still define the value of an s-t cut (A, B) to be the sum of the capacities of all edges e for which the tail of e is in A and the head of e is in B.)

38. You're working with a large database of employee records. For the purposes of this question, we'll picture the database as a two-dimensional table T with a set R of m rows and a set C of n columns; the rows correspond to individual employees, and the columns correspond to different attributes.

To take a simple example, we may have four columns labeled

`name`, `phone number`, `start date`, `manager's name`

and a table with five employees as shown here.

name	phone number	start date	manager's name
Alanis	3-4563	6/13/95	Chelsea
Chelsea	3-2341	1/20/93	Lou
Elrond	3-2345	12/19/01	Chelsea
Hal	3-9000	1/12/97	Chelsea
Raj	3-3453	7/1/96	Chelsea

Given a subset S of the columns, we can obtain a new, smaller table by keeping only the entries that involve columns from S. We will call this new table the *projection* of T onto S, and denote it by $T[S]$. For example, if $S = \{$`name`, `start date`$\}$, then the projection $T[S]$ would be the table consisting of just the first and third columns.

There's a different operation on tables that is also useful, which is to *permute* the columns. Given a permutation p of the columns, we can obtain a new table of the same size as T by simply reordering the columns according to p. We will call this new table the *permutation* of T by p, and denote it by T_p.

All of this comes into play for your particular application, as follows. You have k different subsets of the columns $S_1, S_2, \ldots, S_k$ that you're

going to be working with a lot, so you'd like to have them available in a readily accessible format. One choice would be to store the k projections $T[S_1], T[S_2], \ldots, T[S_k]$, but this would take up a lot of space. In considering alternatives to this, you learn that you may not need to explicitly project onto each subset, because the underlying database system can deal with a subset of the columns particularly efficiently if (in some order) the members of the subset constitute a *prefix* of the columns in left-to-right order. So, in our example, the subsets {`name`, `phone number`} and {`name`, `start date`, `phone number`,} constitute prefixes (they're the first two and first three columns from the left, respectively); and as such, they can be processed much more efficiently in this table than a subset such as {`name`, `start date`}, which does not constitute a prefix. (Again, note that a given subset S_i does not come with a specified order, and so we are interested in whether there is *some* order under which it forms a prefix of the columns.)

So here's the question: Given a parameter $\ell < k$, can you find ℓ permutations of the columns $p_1, p_2, \ldots, p_\ell$ so that for every one of the given subsets S_i (for $i = 1, 2, \ldots, k$), it's the case that the columns in S_i constitute a prefix of at least one of the permuted tables $T_{p_1}, T_{p_2}, \ldots, T_{p_\ell}$? We'll say that such a set of permutations constitutes a valid solution to the problem; if a valid solution exists, it means you only need to store the ℓ permuted tables rather than all k projections. Give a polynomial-time algorithm to solve this problem; for instances on which there is a valid solution, your algorithm should return an appropriate set of ℓ permutations.

Example. Suppose the table is as above, the given subsets are

$$S_1 = \{\texttt{name, phone number}\},$$
$$S_2 = \{\texttt{name, start date}\},$$
$$S_3 = \{\texttt{name, manager's name, start date}\},$$

and $\ell = 2$. Then there is a valid solution to the instance, and it could be achieved by the two permutations

$$p_1 = \{\texttt{name, phone number, start date, manager's name}\},$$
$$p_2 = \{\texttt{name, start date, manager's name, phone number}\}.$$

This way, S_1 constitutes a prefix of the permuted table T_{p_1}, and both S_2 and S_3 constitute prefixes of the permuted table T_{p_2}.

39. You are consulting for an environmental statistics firm. They collect statistics and publish the collected data in a book. The statistics are about populations of different regions in the world and are recorded in

multiples of one million. Examples of such statistics would look like the following table.

Country	A	B	C	Total
grown-up men	11.998	9.083	2.919	24.000
grown-up women	12.983	10.872	3.145	27.000
children	1.019	2.045	0.936	4.000
Total	26.000	22.000	7.000	55.000

We will assume here for simplicity that our data is such that all row and column sums are integers. The Census Rounding Problem is to round all data to integers without changing any row or column sum. Each fractional number can be rounded either up or down. For example, a good rounding for our table data would be as follows.

Country	A	B	C	Total
grown-up men	11.000	10.000	3.000	24.000
grown-up women	13.000	10.000	4.000	27.000
children	2.000	2.000	0.000	4.000
Total	26.000	22.000	7.000	55.000

(a) Consider first the special case when all data are between 0 and 1. So you have a matrix of fractional numbers between 0 and 1, and your problem is to round each fraction that is between 0 and 1 to either 0 or 1 without changing the row or column sums. Use a flow computation to check if the desired rounding is possible.

(b) Consider the Census Rounding Problem as defined above, where row and column sums are integers, and you want to round each fractional number α to either $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. Use a flow computation to check if the desired rounding is possible.

(c) Prove that the rounding we are looking for in (a) and (b) always exists.

40. In a lot of numerical computations, we can ask about the "stability" or "robustness" of the answer. This kind of question can be asked for combinatorial problems as well; here's one way of phrasing the question for the Minimum Spanning Tree Problem.

Suppose you are given a graph $G = (V, E)$, with a cost c_e on each edge e. We view the costs as quantities that have been measured experimentally, subject to possible errors in measurement. Thus, the minimum spanning

tree one computes for G may not in fact be the "real" minimum spanning tree.

Given error parameters $\varepsilon > 0$ and $k > 0$, and a specific edge $e' = (u, v)$, you would like to be able to make a claim of the following form.

> *(∗) Even if the cost of* each *edge were to be changed by at most ε (either increased or decreased), and the costs of k of the edges* other than *e' were further changed to arbitrarily different values, the edge e' would still not belong to any minimum spanning tree of G.*

Such a property provides a type of guarantee that e' is not likely to belong to the minimum spanning tree, even assuming significant measurement error.

Give a polynomial-time algorithm that takes G, e', ε, and k, and decides whether or not property $(*)$ holds for e'.

41. Suppose you're managing a collection of processors and must schedule a sequence of jobs over time.

The jobs have the following characteristics. Each job j has an arrival time a_j when it is first available for processing, a length ℓ_j which indicates how much processing time it needs, and a deadline d_j by which it must be finished. (We'll assume $0 < \ell_j \leq d_j - a_j$.) Each job can be run on any of the processors, but only on one at a time; it can also be preempted and resumed from where it left off (possibly after a delay) on another processor.

Moreover, the collection of processors is not entirely static either: You have an overall pool of k possible processors; but for each processor i, there is an interval of time $[t_i, t_i']$ during which it is available; it is unavailable at all other times.

Given all this data about job requirements and processor availability, you'd like to decide whether the jobs can all be completed or not. Give a polynomial-time algorithm that either produces a schedule completing all jobs by their deadlines or reports (correctly) that no such schedule exists. You may assume that all the parameters associated with the problem are integers.

Example. Suppose we have two jobs J_1 and J_2. J_1 arrives at time 0, is due at time 4, and has length 3. J_2 arrives at time 1, is due at time 3, and has length 2. We also have two processors P_1 and P_2. P_1 is available between times 0 and 4; P_2 is available between times 2 and 3. In this case, there is a schedule that gets both jobs done.

- At time 0, we start job J_1 on processor P_1.

- At time 1, we preempt J_1 to start J_2 on P_1.
- At time 2, we resume J_1 on P_2. (J_2 continues processing on P_1.)
- At time 3, J_2 completes by its deadline. P_2 ceases to be available, so we move J_1 back to P_1 to finish its remaining one unit of processing there.
- At time 4, J_1 completes its processing on P_1.

Notice that there is no solution that does not involve preemption and moving of jobs.

42. Give a polynomial-time algorithm for the following minimization analogue of the Maximum-Flow Problem. You are given a directed graph $G = (V, E)$, with a source $s \in V$ and sink $t \in V$, and numbers (capacities) $\ell(v, w)$ for each edge $(v, w) \in E$. We define a flow f, and the value of a flow, as usual, requiring that all nodes except s and t satisfy flow conservation. However, the given numbers are lower bounds on edge flow—that is, they require that $f(v, w) \geq \ell(v, w)$ for every edge $(v, w) \in E$, and there is no upper bound on flow values on edges.

(a) Give a polynomial-time algorithm that finds a feasible flow of minimum possible value.

(b) Prove an analogue of the Max-Flow Min-Cut Theorem for this problem (i.e., does min-flow = max-cut?).

43. You are trying to solve a circulation problem, but it is not feasible. The problem has demands, but no capacity limits on the edges. More formally, there is a graph $G = (V, E)$, and demands d_v for each node v (satisfying $\sum_{v \in V} d_v = 0$), and the problem is to decide if there is a flow f such that $f(e) \geq 0$ and $f^{in}(v) - f^{out}(v) = d_v$ for all nodes $v \in V$. Note that this problem can be solved via the circulation algorithm from Section 7.7 by setting $c_e = +\infty$ for all edges $e \in E$. (Alternately, it is enough to set c_e to be an extremely large number for each edge—say, larger than the total of all positive demands d_v in the graph.)

You want to fix up the graph to make the problem feasible, so it would be very useful to know why the problem is not feasible as it stands now. On a closer look, you see that there is a subset U of nodes such that there is no edge into U, and yet $\sum_{v \in U} d_v > 0$. You quickly realize that the existence of such a set immediately implies that the flow cannot exist: The set U has a positive total demand, and so needs incoming flow, and yet U has no edges into it. In trying to evaluate how far the problem is from being solvable, you wonder how big the demand of a set with no incoming edges can be.

Give a polynomial-time algorithm to find a subset $S \subset V$ of nodes such that there is no edge into S and for which $\sum_{v \in S} d_v$ is as large as possible subject to this condition.

44. Suppose we are given a directed network $G = (V, E)$ with a root node r and a set of *terminals* $T \subseteq V$. We'd like to disconnect many terminals from r, while cutting relatively few edges.

We make this trade-off precise as follows. For a set of edges $F \subseteq E$, let $q(F)$ denote the number of nodes $v \in T$ such that there is no r-v path in the subgraph $(V, E - F)$. Give a polynomial-time algorithm to find a set F of edges that maximizes the quantity $q(F) - |F|$. (Note that setting F equal to the empty set is an option.)

45. Consider the following definition. We are given a set of n countries that are engaged in trade with one another. For each country i, we have the value s_i of its budget surplus; this number may be positive or negative, with a negative number indicating a deficit. For each pair of countries i, j, we have the total value e_{ij} of all exports from i to j; this number is always nonnegative. We say that a subset S of the countries is *free-standing* if the sum of the budget surpluses of the countries in S, minus the total value of all exports from countries in S to countries not in S, is nonnegative.

Give a polynomial-time algorithm that takes this data for a set of n countries and decides whether it contains a nonempty free-standing subset that is not equal to the full set.

46. In sociology, one often studies a graph G in which nodes represent people and edges represent those who are friends with each other. Let's assume for purposes of this question that friendship is symmetric, so we can consider an undirected graph.

Now suppose we want to study this graph G, looking for a "close-knit" group of people. One way to formalize this notion would be as follows. For a subset S of nodes, let $e(S)$ denote the number of edges in S—that is, the number of edges that have both ends in S. We define the *cohesiveness* of S as $e(S)/|S|$. A natural thing to search for would be a set S of people achieving the maximum cohesiveness.

(a) Give a polynomial-time algorithm that takes a rational number α and determines whether there exists a set S with cohesiveness at least α.

(b) Give a polynomial-time algorithm to find a set S of nodes with maximum cohesiveness.

47. The goal of this problem is to suggest variants of the Preflow-Push Algorithm that speed up the practical running time without ruining its worst-case complexity. Recall that the algorithm maintains the invariant that $h(v) \leq h(w) + 1$ for all edges (v, w) in the residual graph of the current preflow. We proved that if f is a flow (not just a preflow) with this invariant, then it is a maximum flow. Heights were monotone increasing, and the running-time analysis depended on bounding the number of times nodes can increase their heights. Practical experience shows that the algorithm is almost always much faster than suggested by the worst case, and that the practical bottleneck of the algorithm is relabeling nodes (and not the nonsaturating pushes that lead to the worst case in the theoretical analysis). The goal of the problems below is to decrease the number of relabelings by increasing heights faster than one by one. Assume you have a graph G with n nodes, m edges, capacities c, source s, and sink t.

(a) The Preflow-Push Algorithm, as described in Section 7.4, starts by setting the flow equal to the capacity c_e on all edges e leaving the source, setting the flow to 0 on all other edges, setting $h(s) = n$, and setting $h(v) = 0$ for all other nodes $v \in V$. Give an $O(m)$ procedure for initializing node heights that is better than the one we constructed in Section 7.4. Your method should set the height of each node v to be as high as possible given the initial flow.

(b) In this part we will add a new step, called *gap relabeling*, to Preflow-Push, which will increase the labels of lots of nodes by more than one at a time. Consider a preflow f and heights h satisfying the invariant. A *gap* in the heights is an integer $0 < h < n$ so that no node has height exactly h. Assume h is a gap value, and let A be the set of nodes v with heights $n > h(v) > h$. Gap relabeling is the process of changing the heights of all nodes in A so they are equal to n. Prove that the Preflow-Push Algorithm with gap relabeling is a valid max-flow algorithm. Note that the only new thing that you need to prove is that gap relabeling preserves the invariant above, that $h(v) \leq h(w) + 1$ for all edges (v, w) in the residual graph.

(c) In Section 7.4 we proved that $h(v) \leq 2n - 1$ throughout the algorithm. Here we will have a variant that has $h(v) \leq n$ throughout. The idea is that we "freeze" all nodes when they get to height n; that is, nodes at height n are no longer considered active, and hence are not used for `push` and `relabel`. This way, at the end of the algorithm we have a preflow and height function that satisfies the invariant above, and so that all excess is at height n. Let B be the set of nodes v so that there

is a path from v to t in the residual graph of the current preflow. Let $A = V - B$. Prove that at the end of the algorithm, (A, B) is a minimum-capacity s-t cut.

(d) The algorithm in part (c) computes a minimum s-t cut but fails to find a maximum flow (as it ends with a preflow that has excesses). Give an algorithm that takes the preflow f at the end of the algorithm of part (c) and converts it into a maximum flow in at most $O(mn)$ time. (*Hint:* Consider nodes with excess, and try to send the excess back to s using only edges that the flow came on.)

48. In Section 7.4 we considered the Preflow-Push Algorithm, and discussed one particular selection rule for considering vertices. Here we will explore a different selection rule. We will also consider variants of the algorithm that terminate early (and find a cut that is close to the minimum possible).

(a) Let f be any preflow. As f is not necessarily a valid flow, it is possible that the value $f^{out}(s)$ is much higher than the maximum-flow value in G. Show, however, that $f^{in}(t)$ is a lower bound on the maximum-flow value.

(b) Consider a preflow f and a compatible labeling h. Recall that the set $A = \{v :$ There is an s-v path in the residual graph $G_f\}$, and $B = V - A$ defines an s-t cut for any preflow f that has a compatible labeling h. Show that the capacity of the cut (A, B) is equal to $c(A, B) = \sum_{v \in B} e_f(v)$.

Combining (a) and (b) allows the algorithm to terminate early and return (A, B) as an approximately minimum-capacity cut, assuming $c(A, B) - f^{in}(t)$ is sufficiently small. Next we consider an implementation that will work on decreasing this value by trying to push flow out of nodes that have a lot of excess.

(c) The scaling version of the Preflow-Push Algorithm maintains a scaling parameter Δ. We set Δ initially to be a large power of 2. The algorithm at each step selects a node with excess at least Δ with as small a height as possible. When no nodes (other than t) have excess at least Δ, we divide Δ by 2, and continue. Note that this is a valid implementation of the generic Preflow-Push Algorithm. The algorithm runs in phases. A single phase continues as long as Δ is unchanged. Note that Δ starts out at the largest capacity, and the algorithm terminates when $\Delta = 1$. So there are at most $O(\log C)$ scaling phases. Show how to implement this variant of the algorithm so that the running time can be bounded by $O(mn + n \log C + K)$ if the algorithm has K nonsaturating `push` operations.

(d) Show that the number of nonsaturating `push` operations in the above algorithm is at most $O(n^2 \log C)$. Recall that $O(\log C)$ bounds the number of scaling phases. To bound the number of nonsaturating `push` operations in a single scaling phase, consider the potential function $\Phi = \sum_{v \in V} h(v) e_f(v)/\Delta$. What is the effect of a nonsaturating push on Φ? Which operation(s) can make Φ increase?

49. Consider an assignment problem where we have a set of n stations that can provide service, and there is a set of k requests for service. Say, for example, that the stations are cell towers and the requests are cell phones. Each request can be served by a given set of stations. The problem so far can be represented by a bipartite graph G: one side is the stations, the other the customers, and there is an edge (x, y) between customer x and station y if customer x can be served from station y. Assume that each station can serve at most one customer. Using a max-flow computation, we can decide whether or not all customers can be served, or can get an assignment of a subset of customers to stations maximizing the number of served customers.

Here we consider a version of the problem with an additional complication: Each customer offers a different amount of money for the service. Let U be the set of customers, and assume that customer $x \in U$ is willing to pay $v_x \geq 0$ for being served. Now the goal is to find a subset $X \subset U$ maximizing $\sum_{x \in X} v_x$ such that there is an assignment of the customers in X to stations.

Consider the following greedy approach. We process customers in order of decreasing value (breaking ties arbitrarily). When considering customer x the algorithm will either "promise" service to x or reject x in the following greedy fashion. Let X be the set of customers that so far have been promised service. We add x to the set X if and only if there is a way to assign $X \cup \{x\}$ to servers, and we reject x otherwise. Note that rejected customers will not be considered later. (This is viewed as an advantage: If we need to reject a high-paying customer, at least we can tell him/her early.) However, we do not assign accepted customers to servers in a greedy fashion: we only fix the assignment after the set of accepted customers is fixed. Does this greedy approach produce an optimal set of customers? Prove that it does, or provide a counterexample.

50. Consider the following scheduling problem. There are m machines, each of which can process jobs, one job at a time. The problem is to assign jobs to machines (each job needs to be assigned to exactly one machine) and order the jobs on machines so as to minimize a cost function.

The machines run at different speeds, but jobs are identical in their processing needs. More formally, each machine i has a parameter ℓ_i, and each job requires ℓ_i time if assigned to machine i.

There are n jobs. Jobs have identical processing needs but different levels of urgency. For each job j, we are given a cost function $c_j(t)$ that is the cost of completing job j at time t. We assume that the costs are nonnegative, and monotone in t.

A schedule consists of an assignment of jobs to machines, and on each machine the schedule gives the order in which the jobs are done. The job assigned to machine i as the first job will complete at time ℓ_i, the second job at time $2\ell_i$ and so on. For a schedule S, let $t_S(j)$ denote the completion time of job j in this schedule. The cost of the schedule is $cost(S) = \sum_j c_j(t_S(j))$.

Give a polynomial-time algorithm to find a schedule of minimum cost.

51. Some friends of yours have grown tired of the game "Six Degrees of Kevin Bacon" (after all, they ask, isn't it just breadth-first search?) and decide to invent a game with a little more punch, algorithmically speaking. Here's how it works.

You start with a set X of n actresses and a set Y of n actors, and two players P_0 and P_1. Player P_0 names an actress $x_1 \in X$, player P_1 names an actor y_1 who has appeared in a movie with x_1, player P_0 names an actress x_2 who has appeared in a movie with y_1, and so on. Thus, P_0 and P_1 collectively generate a sequence $x_1, y_1, x_2, y_2, \ldots$ such that each actor/actress in the sequence has costarred with the actress/actor immediately preceding. A player P_i ($i = 0, 1$) loses when it is P_i's turn to move, and he/she cannot name a member of his/her set who hasn't been named before.

Suppose you are given a specific pair of such sets X and Y, with complete information on who has appeared in a movie with whom. A *strategy* for P_i, in our setting, is an algorithm that takes a current sequence $x_1, y_1, x_2, y_2, \ldots$ and generates a legal next move for P_i (assuming it's P_i's turn to move). Give a polynomial-time algorithm that decides which of the two players can force a win, in a particular instance of this game.

Notes and Further Reading

Network flow emerged as a cohesive subject through the work of Ford and Fulkerson (1962). It is now a field of research in itself, and one can easily

devote an entire course to the topic; see, for example, the survey by Goldberg, Tardos, and Tarjan (1990) and the book by Ahuja, Magnanti, and Orlin (1993).

Schrijver (2002) provides an interesting historical account of the early work by Ford and Fulkerson on the flow problem. Lending further support to those of us who always felt that the Minimum-Cut Problem had a slightly destructive overtone, this survey cites a recently declassified U.S. Air Force report to show that in the original motivating application for minimum cuts, the network was a map of rail lines in the Soviet Union, and the goal was to disrupt transportation through it.

As we mention in the text, the formulations of the Bipartite Matching and Disjoint Paths Problems predate the Maximum-Flow Problem by several decades; it was through the development of network flows that these were all placed on a common methodological footing. The rich structure of matchings in bipartite graphs has many independent discoverers; P. Hall (1935) and König (1916) are perhaps the most frequently cited. The problem of finding edge-disjoint paths from a source to a sink is equivalent to the Maximum-Flow Problem with all capacities equal to 1; this special case was solved (in essentially equivalent form) by Menger (1927).

The Preflow-Push Maximum-Flow Algorithm is due to Goldberg (1986), and its efficient implementation is due to Goldberg and Tarjan (1986). High-performance code for this and other network flow algorithms can be found at a Web site maintained by Andrew Goldberg.

The algorithm for image segmentation using minimum cuts is due to Greig, Porteous, and Seheult (1989), and the use of minimum cuts has become an active theme in computer vision research (see, e.g., Veksler (1999) and Kolmogorov and Zabih (2004) for overviews); we will discuss some further extensions of this approach in Chapter 12. Wayne (2001) presents further results on baseball elimination and credits Alan Hoffman with initially popularizing this example in the 1960s. Many further applications of network flows and cuts are discussed in the book by Ahuja, Magnanti, and Orlin (1993).

The problem of finding a minimum-cost perfect matching is a special case of the *Minimum-Cost Flow Problem*, which is beyond the scope of our coverage here. There are a number of equivalent ways to state the Minimum-Cost Flow Problem; in one formulation, we are given a flow network with both capacities c_e and costs C_e on the edges; the *cost* of a flow f is equal to the sum of the edge costs weighted by the amount of flow they carry, $\sum_e C_e f(e)$, and the goal is to produce a maximum flow of minimum total cost. The Minimum-Cost Flow Problem can be solved in polynomial time, and it too has many applications;

Cook et al. (1998) and Ahuja, Magnanti, and Orlin (1993) discuss algorithms for this problem.

While network flow models routing problems that can be reduced to the task of constructing a number of paths from a single source to a single sink, there is a more general, and harder, class of routing problems in which paths must be simultaneously constructed between different pairs of senders and receivers. The relationship among these classes of problems is a bit subtle; we discuss this issue, as well as algorithms for some of these harder types of routing problems, in Chapter 11.

Notes on the Exercises Exercise 8 is based on a problem we learned from Bob Bland; Exercise 16 is based on discussions with Udi Manber; Exercise 25 is based on discussions with Jordan Erenrich; Exercise 35 is based on discussions with Yuri Boykov, Olga Veksler, and Ramin Zabih; Exercise 36 is based on results of Hiroshi Ishikawa and Davi Geiger, and of Boykov, Veksler, and Zabih; Exercise 38 is based on a problem we learned from Al Demers; and Exercise 46 is based on a result of J. Picard and H. Ratliff.