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.

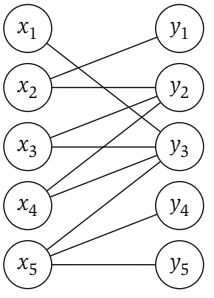

Figure 7.1 A bipartite graph.

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.

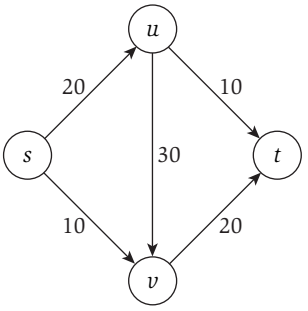

Figure 7.2 A flow network, with source s and sink t. The numbers next to the edges are the capacities.

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

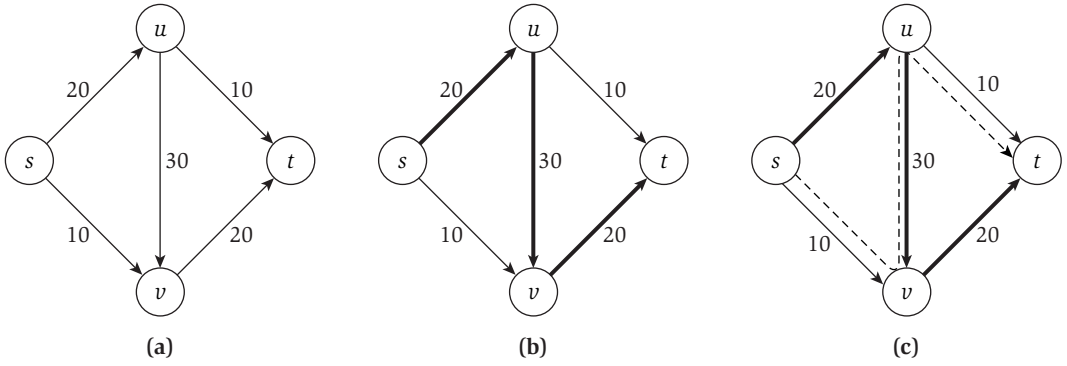

Figure 7.3 (a) The network of Figure 7.2. (b) Pushing 20 units of flow along the path s, u, v, t. (c) The new kind of augmenting path using the edge (u, v) backward.

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.

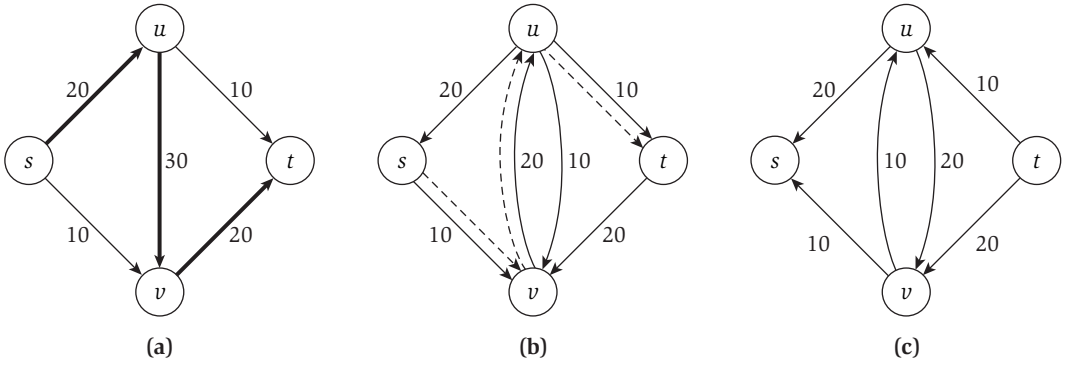

Figure 7.4 (a) The graph G with the path s, u, v, t used to push the first 20 units of flow. (b) The residual graph of the resulting flow f, with the residual capacity next to each edge. The dotted line is the new augmenting path. (c) The residual graph after pushing an additional 10 units of flow along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

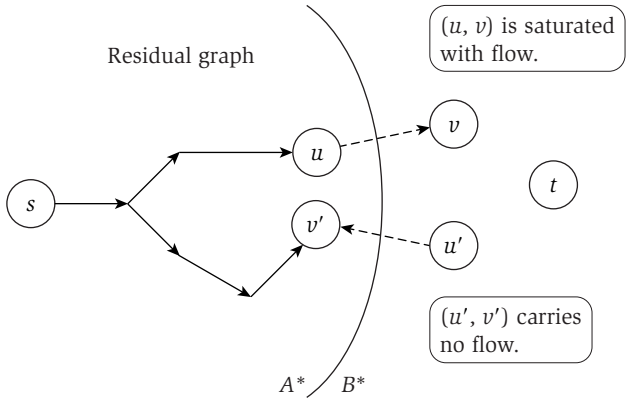

Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

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*

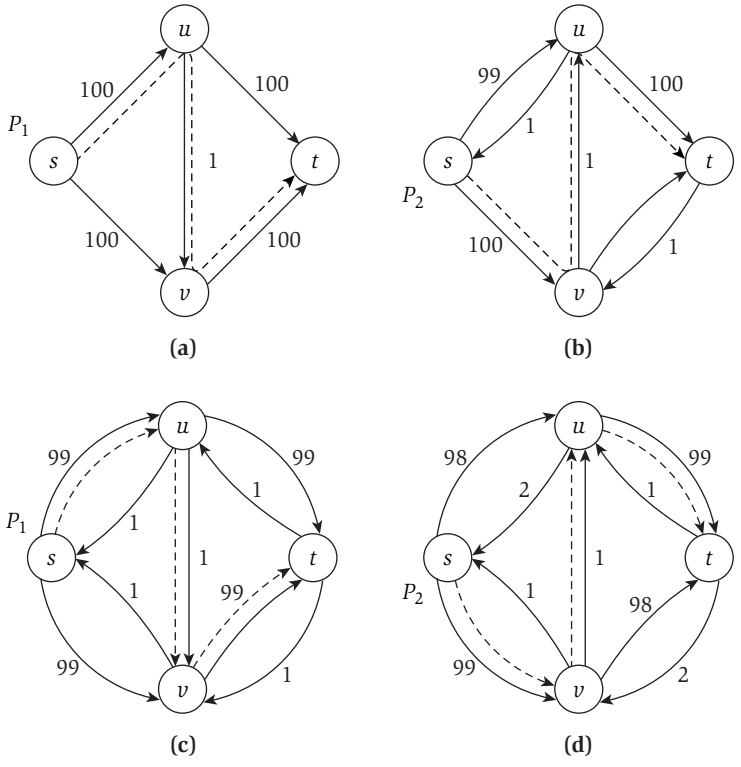

Figure 7.6 Parts (a) through (d) depict four iterations of the Ford-Fulkerson Algorithm using a bad choice of augmenting paths: The augmentations alternate between the path P_1 through the nodes s, u, v, t in order and the path P_2 through the nodes s, v, u, t in order.

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 $h(v) \leq h(w) + 1$.

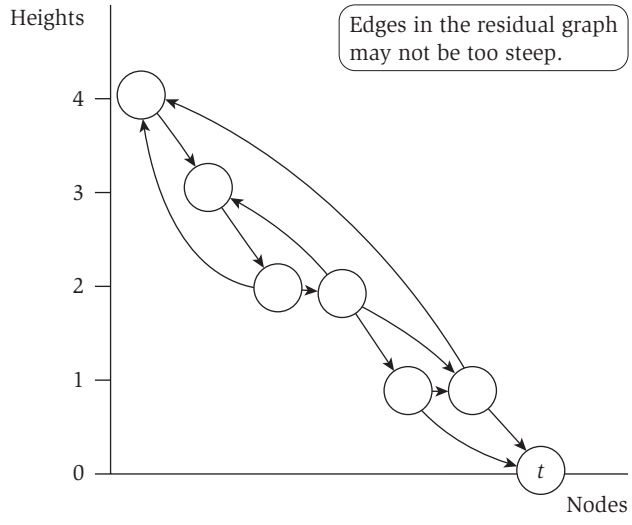

Figure 7.7 A residual graph and a compatible labeling. No edge in the residual graph can be too "steep"—its tail can be at most one unit above its head in height. The source node s must have $h(s) = n$ and is not drawn in the figure.

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.

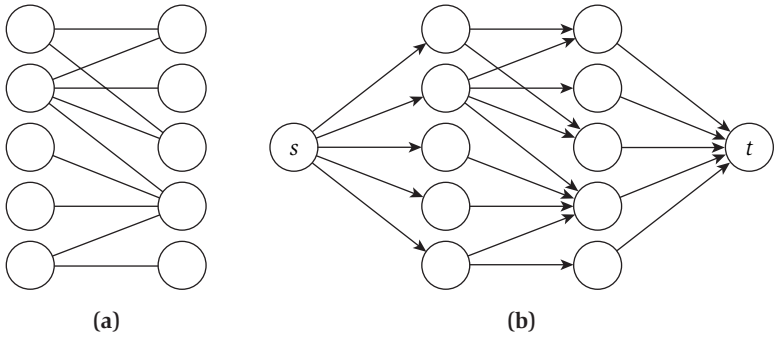

Figure 7.9 (a) A bipartite graph. (b) The corresponding flow network, with all capacities equal to 1.

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.

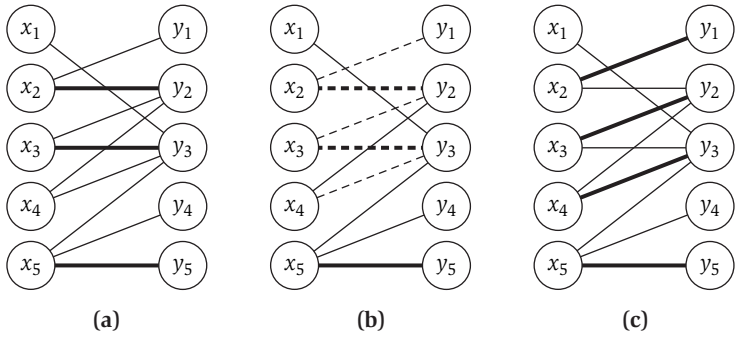

Figure 7.10 (a) A bipartite graph, with a matching M. (b) The augmenting path in the corresponding residual graph. (c) The matching obtained by the augmentation.

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

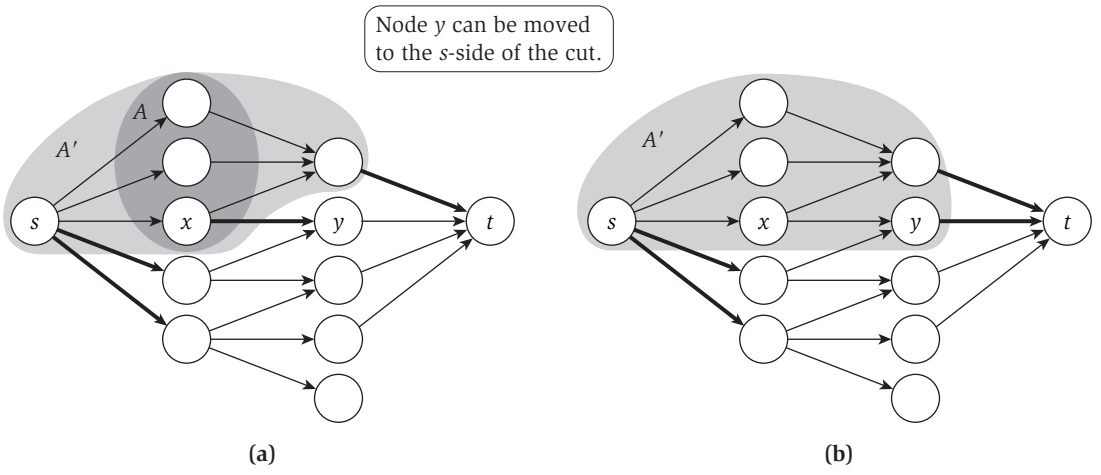

Figure 7.11 (a) A minimum cut in proof of (7.40). (b) The same cut after moving node y to the A' side. The edges crossing the cut are dark.

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. ■