have been added to this basic approach to shortest paths; a Web site maintained by Andrew Goldberg contains state-of-the-art code that he has developed for this problem (among a number of others), based on work by Cherkassky, Goldberg and Radzik (1994). The applications of shortest-path methods to Internet routing, and the trade-offs among the different algorithms for networking applications, are covered in books by Bertsekas and Gallager (1992), Keshav (1997), and Stewart (1998).

Notes on the Exercises Exercise 5 is based on discussions with Lillian Lee; Exercise 6 is based on a result of Donald Knuth; Exercise 25 is based on results of Dimitris Bertsimas and Andrew Lo; and Exercise 29 is based on a result of S. Dreyfus and R. Wagner.

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.

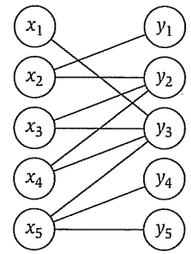

Figure 7.1 A bipartite graph.

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.

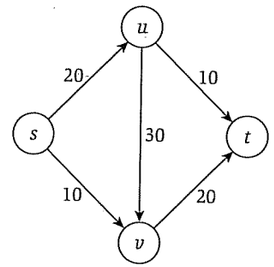

Figure 7.2 A flow network, with source s and sink t. The numbers next to the edges are the capacities.

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 $0 \le f(e) \le c_e$.

(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

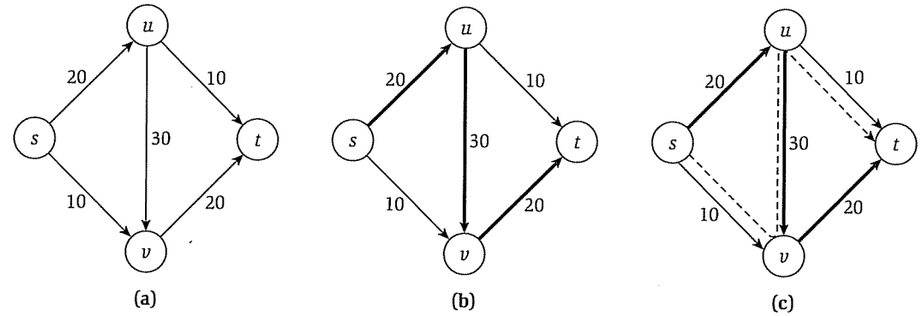

Figure 7.3 (a) The network of Figure 7.2. (b) Pushing 20 units of flow along the path s, u, v, t. (c) The new kind of augmenting path using the edge (u, v) backward.

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.

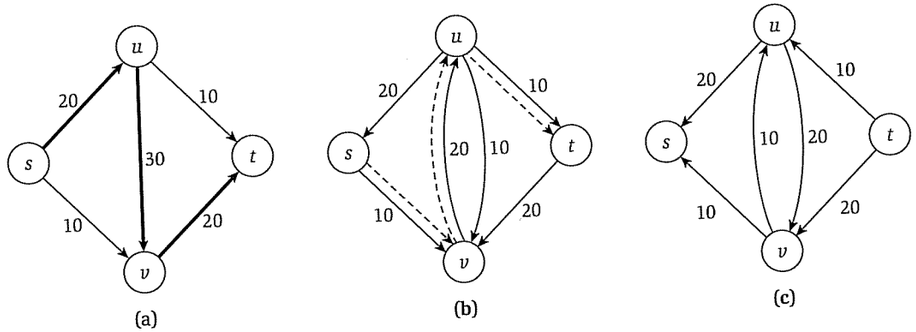

Figure 7.4 (a) The graph G with the path s, u, v, t used to push the first 20 units of flow. (b) The residual graph of the resulting flow f, with the residual capacity next to each edge. The dotted line is the new augmenting path. (c) The residual graph after pushing an additional 10 units of flow along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

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

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

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

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 $\texttt{bottleneck}(P, f)$ 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 $v(f') = v(f) + \texttt{bottleneck}(P, f)$; and since $\texttt{bottleneck}(P, f) > 0$, 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 $\texttt{bottleneck}(P, f)$, 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 $\texttt{bottleneck}(P, f)$. ■

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 $v(f) \leq c(A, B)$.*

Referrence:

Text Book:

Algorithm Design, Jon Kleinberg and Eva Tardos, Pearson New International Edition.