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.

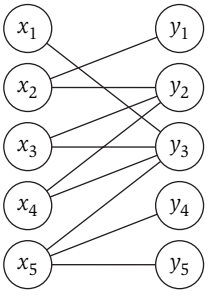

Figure 7.1 A bipartite graph.

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.

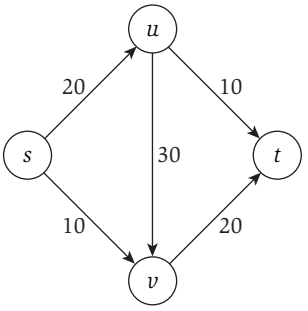

Figure 7.2 A flow network, with source s and sink t. The numbers next to the edges are the capacities.

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

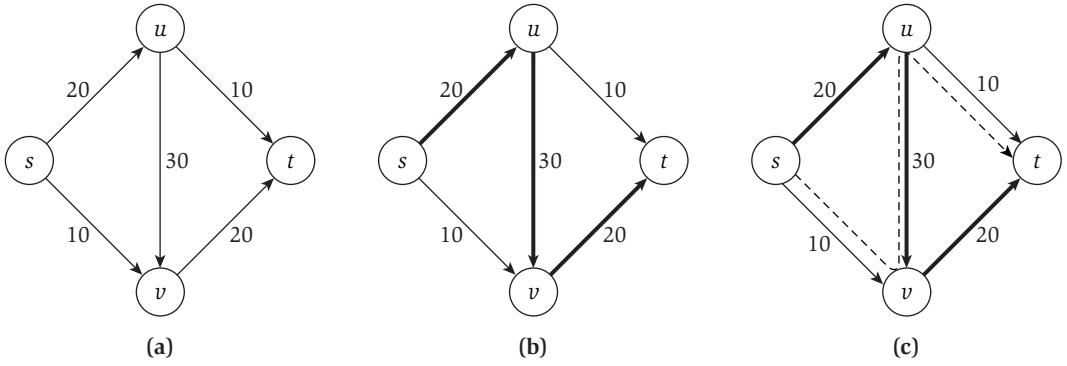

Figure 7.3 (a) The network of Figure 7.2. (b) Pushing 20 units of flow along the path s, u, v, t. (c) The new kind of augmenting path using the edge (u, v) backward.

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.

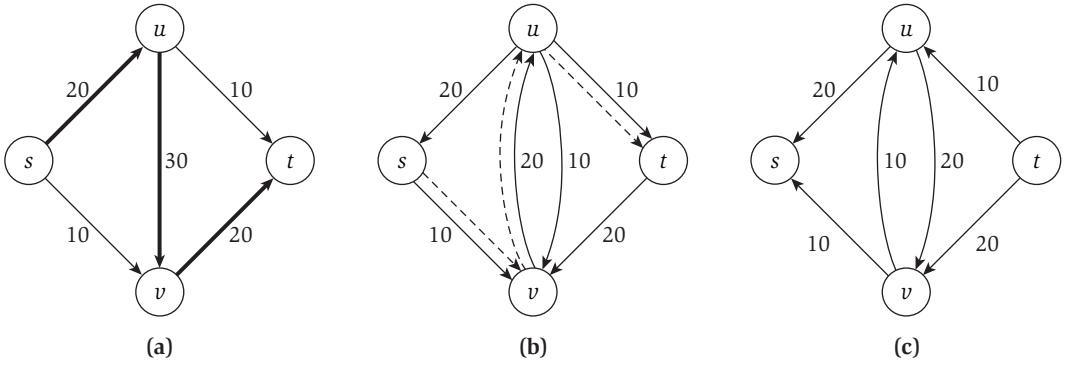

Figure 7.4 (a) The graph G with the path s, u, v, t used to push the first 20 units of flow. (b) The residual graph of the resulting flow f, with the residual capacity next to each edge. The dotted line is the new augmenting path. (c) The residual graph after pushing an additional 10 units of flow along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