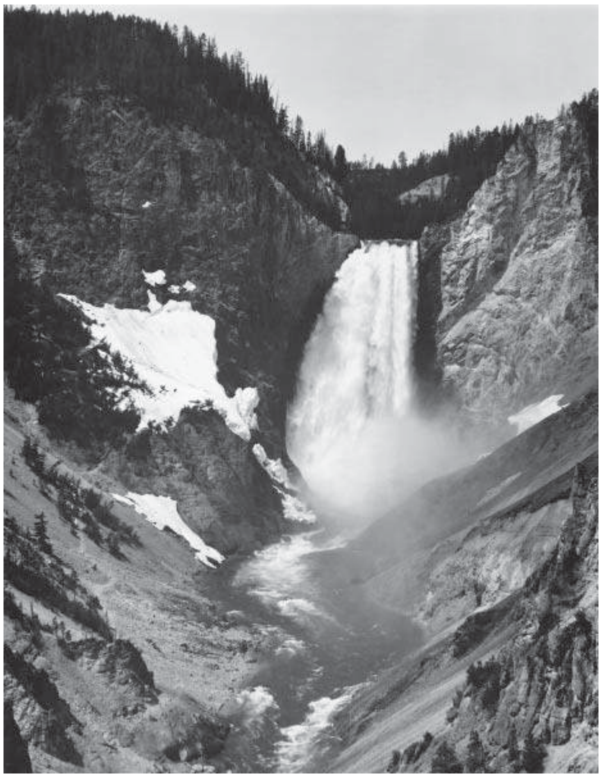
# Chapter 16 Network Flow and Matching


Yellowstone Falls, 1941. Ansel Adams. U.S. government image. U.S. National Archives and Records Administration.

## Contents

Consider a computer network modeled by a directed graph $G$ in which each vertex represents a computer, each edge $(u, v)$ represents a one-way communication channel from computer $u$ to computer $v$, and the weight of each edge $(u, v)$ represents the bandwidth of the channel, that is, the maximum number of bytes that can be sent from $u$ to $v$ in one second. Suppose that we would like to send a high-bandwidth streaming media connection from some computer $s$ in $G$ to some computer $t$ in $G$, with as high a bandwidth as possible, possibly even higher than the maximum bandwidth of any single link in our network. This might seem impossible at first, but it might actually be possible if we can divide this media stream into lots of packets and route these packets through multiple paths in the network.

We can formulate this problem by imagining that each edge in $G$ represents a "pipe" that can transport some commodity, with the weight of that edge representing the maximum amount it can transport per unit time interval. The optimization problem is then known as the ***maximum flow*** problem, where we are given a weighted directed graph and asked to find a way of transporting the maximum amount of the given commodity from some vertex $s$, called the ***source***, to some vertex $t$, called the ***sink***. (See Figure 16.1.)

Incidentally, the maximum flow problem is closely related to the problem of finding the maximum way of matching vertices of one type in a graph with vertices of another type. We therefore also study the maximum matching problem, showing how the maximum flow problem can be used to solve it efficiently.

Sometimes we have many different maximum flows. Although all are maximum in terms of how much flow they produce, these flows may in fact be different in how much they cost. Thus, in this chapter, we also study methods for computing maximum flows that are of minimum cost, when there are many different maximum flows and we have some way of measuring their relative costs.

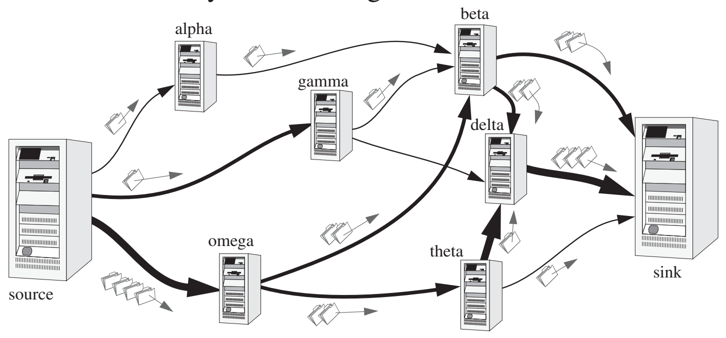


**Figure 16.1:** A flow in a graph representing a computer network, with the bandwidth of thick edges being 4 MB/s, medium edges being 2 MB/s, and thin edges being 1 MB/s. We indicate the flow using icons, where each folder corresponds to one MB/s going through the channel. Note that the total amount of flow sent from the source to the sink (6 MB/s) is not maximum. Indeed, one additional MB/s can be pushed from the source to gamma, from gamma to delta, and from delta to the sink. After this extra flow is added, the total flow will be maximum.

## 16.1 Flows and Cuts

The above example illustrates the rules that a legal flow must obey. In order to precisely say what these rules are, let us carefully define what we mean by a flow.

### Flow Networks

A ***flow network*** $N$ consists of the following:

- A connected directed graph $G$ with nonnegative integer weights on the edges, where the weight of an edge $e$ is called the ***capacity*** $c(e)$ of $e$
- Two distinguished vertices, $s$ and $t$, of $G$, called the ***source*** and ***sink***, respectively, such that $s$ has no incoming edges and $t$ has no outgoing edges.

Given such a labeled graph, the challenge is to determine the maximum amount of some commodity that can be pushed from $s$ to $t$ under the constraint that the capacity of an edge determines the maximum flow that can go along that edge. (See Figure 16.2.)

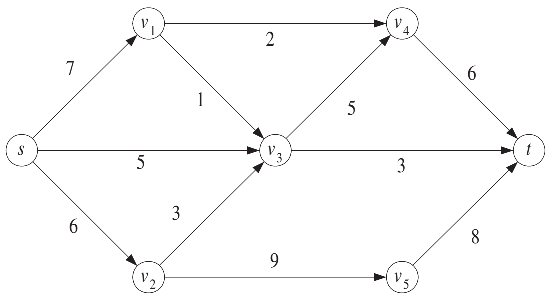


**Figure 16.2:** A flow network, $N$. Each edge $e$ of $N$ is labeled with its capacity, $c(e)$.

Of course, if we wish some commodity to flow from $s$ to $t$, we need to be more precise about what we mean by a "flow." A ***flow*** for network $N$ is an assignment of an integer value $f(e)$ to each edge $e$ of $G$ that satisfies the following properties:

- For each edge $e$ of $G$,
$$0 \leq f(e) \leq c(e) \quad (\textbf{\textit{capacity rule}}).$$
- For each vertex $v$ of $G$ distinct from the source $s$ and the sink $t$
$$\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e) \quad (\textbf{\textit{conservation rule}}),$$
where $E^-(v)$ and $E^+(v)$ denote the sets of incoming and outgoing edges of $v$, respectively.

In other words, a flow must satisfy the edge capacity constraints and must, for every vertex $v$ other than $s$ and $t$, have the total amount of flow going out of $v$ equal to the total amount of flow coming into $v$. Each of the above rules is satisfied, for example, by the flow illustrated in Figure 16.3.

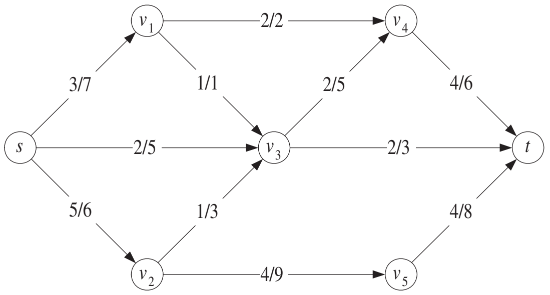


**Figure 16.3:** A flow, $f$ (of value $|f| = 10$), for the flow network $N$ of Figure 16.2.

The quantity $f(e)$ is called the ***flow*** of edge $e$. The ***value*** of a flow $f$, which we denote by $|f|$, is equal to the total amount of flow coming out from the source $s$:

$$|f| = \sum_{e \in E^+(s)} f(e).$$

It is easy to show that the flow value is also equal to the total amount of flow going into the sink $t$ (see Exercise R-16.1):

$$|f| = \sum_{e \in E^-(t)} f(e).$$

That is, a flow specifies how some commodity is pushed out from $s$, through the network $N$, and finally into the sink $t$. A ***maximum flow*** for flow network $N$ is a flow with maximum value over all flows for $N$ (see Figure 16.4). Since a maximum flow is using a flow network most efficiently, we are most interested in methods for computing maximum flows.

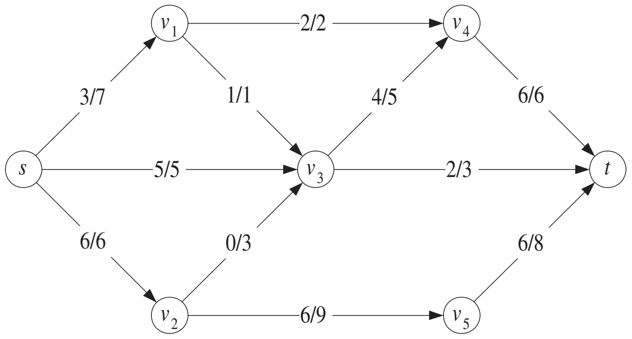
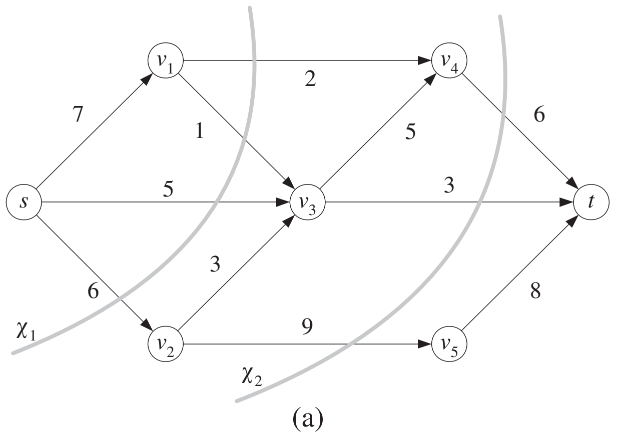


**Figure 16.4:** A maximum flow $f^*$ (of value $|f^*| = 14$) for the flow network $N$ of Figure 16.2.

### 16.1.1 Cuts

It turns out that flows are closely related to another concept, known as cuts. Intuitively, a cut is a division of the vertices of a flow network $N$ into two sets, with $s$ on one side and $t$ on the other. Formally, a ***cut*** of $N$ is a partition $\chi = (V_s, V_t)$ of the vertices of $N$ such that $s \in V_s$ and $t \in V_t$. An edge $e$ of $N$ with origin $u \in V_s$ and destination $v \in V_t$ is said to be a ***forward edge*** of cut $\chi$. An edge with origin in $V_t$ and destination in $V_s$ is said to be a ***backward edge***. We envision a cut as a separation of $s$ and $t$ in $N$ done by cutting across edges of $N$, with forward edges going from $s$'s side to $t$'s side and backward edges going in the opposite direction. (See Figure 16.5.)



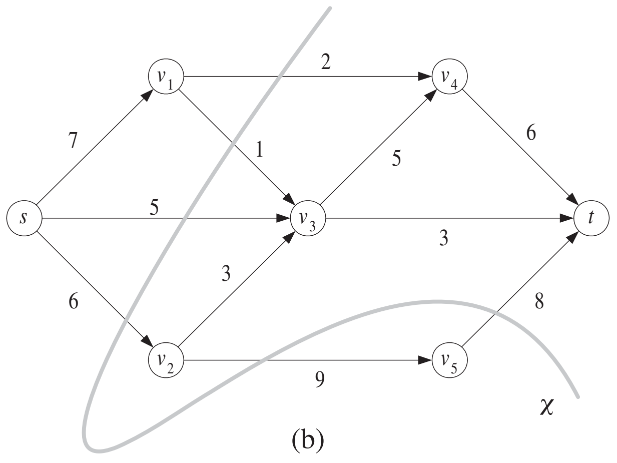


**Figure 16.5:** (a) Two cuts, $\chi_1$ (on the left) and $\chi_2$ (on the right), in the flow network $N$ of Figure 16.2. These cuts have only forward edges and their capacities are $c(\chi_1) = 14$ and $c(\chi_2) = 18$. Cut $\chi_1$ is a minimum cut for $N$. (b) A cut $\chi$ in $N$ with both forward and backward edges. Its capacity is $c(\chi) = 22$.

Given a flow $f$ for $N$, the ***flow across cut*** $\chi$, denoted $f(\chi)$, is equal to the sum of the flows in the forward edges of $\chi$ minus the sum of the flows in the backward edges of $\chi$. That is, $f(\chi)$ is the net amount of commodity that flows from $s$'s side of $\chi$ to $t$'s side of $\chi$. The following lemma shows an interesting property of $f(\chi)$.

**Lemma 16.1:** *Let $N$ be a flow network, and let $f$ be a flow for $N$. For any cut $\chi$ of $N$, the value of $f$ is equal to the flow across cut $\chi$, that is, $|f| = f(\chi)$.*

**Proof:** Consider the sum

$$F = \sum_{v \in V_s} \left( \sum_{e \in E^+(v)} f(e) - \sum_{e \in E^-(v)} f(e) \right).$$

By the conservation rule, for each vertex $v$ of $V_s$ distinct from $s$, we have that $\sum_{e \in E^+(v)} f(e) - \sum_{e \in E^-(v)} f(e) = 0$. Thus, $F = |f|$.

On the other hand, for each edge $e$ that is not a forward or a backward edge of cut $\chi$, the sum $F$ contains both the term $f(e)$ and the term $-f(e)$, which cancel each other, or neither the term $f(e)$ nor the term $-f(e)$. Thus, $F = f(\chi)$. ■

The above theorem shows that no matter where we cut a flow network to separate $s$ and $t$, the flow across that cut is equal to the flow for the entire network. The ***capacity*** of cut $\chi$, denoted $c(\chi)$, is the sum of the capacities of the forward edges of $\chi$ (note that we do not include the backward edges). The next lemma shows that a cut capacity $c(\chi)$ is an upper bound on any flow across $\chi$.

**Lemma 16.2:** *Let $N$ be a flow network, and let $\chi$ be a cut of $N$. Given any flow $f$ for $N$, the flow across cut $\chi$ does not exceed the capacity of $\chi$, that is, $f(\chi) \le c(\chi)$.*

**Proof:** Denote with $E^+(\chi)$ the forward edges of $\chi$, and with $E^-(\chi)$ the backward edges of $\chi$. By the definition of $f(\chi)$, we have

$$f(\chi) = \sum_{e \in E^+(\chi)} f(e) - \sum_{e \in E^-(\chi)} f(e).$$

Dropping nonpositive terms from the above sum, we obtain the simplified condition, $f(\chi) \le \sum_{e \in E^+(\chi)} f(e)$. By the capacity rule, for each edge $e$, $f(e) \le c(e)$. Thus, we have

$$f(\chi) \le \sum_{e \in E^+(\chi)} c(e) = c(\chi).$$

■

By combining Lemmas 16.1 and 16.2, we obtain the following important result relating flows and cuts.

**Theorem 16.3:** *Let $N$ be a flow network. Given any flow $f$ for $N$ and any cut $\chi$ of $N$, the value of $f$ does not exceed the capacity of $\chi$, that is, $|f| \le c(\chi)$.*

In other words, given any cut $\chi$ for a flow network $N$, the capacity of $\chi$ is an upper bound on any flow for $N$. This upper bound holds even for a ***minimum cut*** of

$N$, which is a cut with minimum capacity, taken over all cuts of $N$. In the example of Figure 16.5, $\chi_1$ is a minimum cut.

## 16.1.2 Residual Capacity and Augmenting Paths

Theorem 16.3 implies that the value of a maximum flow is no more than the capacity of a minimum cut. We will show in this section that these two quantities are actually equal. In the process, we will outline an approach for constructing a maximum flow.

### Residual Capacity

In order to prove that a certain flow $f$ is maximum, we need some way of showing that there is absolutely no more flow that can possibly be "squeezed" into $f$. Using the related concepts of residual capacity and augmenting paths, discussed next, we can provide just such a proof for when a flow $f$ is maximum.

Let $N$ be a flow network, which is specified by a graph $G$, capacity function $c$, source $s$, and sink $t$. Furthermore, let $f$ be a flow for $N$. Given an edge $e$ of $G$ directed from vertex $u$ to vertex $v$, the ***residual capacity*** from $u$ to $v$ with respect to the flow $f$, denoted $\Delta_f(u,v)$, is defined as

$$\Delta_f(u,v) = c(e) - f(e),$$

and the residual capacity from $v$ to $u$ is defined as

$$\Delta_f(v,u) = f(e).$$

Intuitively, the residual capacity defined by a flow $f$ is any additional capacity that $f$ has not fully taken advantage of in "pushing" its flow from $s$ to $t$.

Let $\pi$ be a path from $s$ to $t$ that is allowed to traverse edges in either the forward or backward direction, that is, we can traverse the edge $e = (u,v)$ from its origin $u$ to its destination $v$ or from its destination $v$ to its origin $u$. Formally, a ***forward edge*** of $\pi$ is an edge $e$ of $\pi$ such that, in going from $s$ to $t$ along path $\pi$, the origin of $e$ is encountered before the destination of $e$. An edge of $\pi$ that is not forward is said to be a ***backward edge***. Let us extend our definition of ***residual capacity*** to an edge $e$ in $\pi$ traversed from $u$ to $v$, so that $\Delta_f(e) = \Delta_f(u,v)$. In other words,

$$\Delta_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \text{ is a forward edge} \\ f(e) & \text{if } e \text{ is a backward edge.} \end{cases}$$

That is, the residual capacity of an edge $e$ going in the forward direction is the additional capacity of $e$ that $f$ has yet to consume, but the residual capacity in the opposite direction is the flow that $f$ has consumed (and could potentially "give back" if that allows for another flow of higher value).

### Augmenting Paths

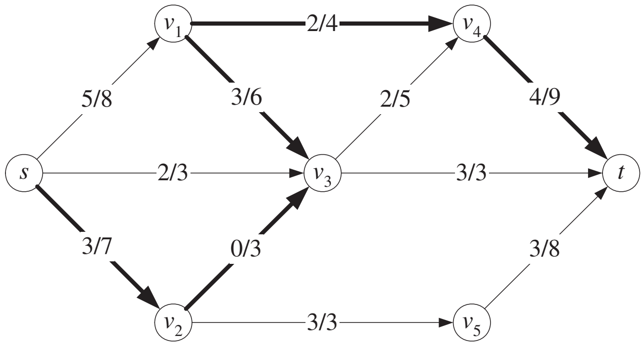
The residual capacity, $\Delta_f(\pi)$, of a path $\pi$ is the minimum residual capacity of its edges. That is,

$$\Delta_f(\pi) = \min_{e \in \pi} \Delta_f(e).$$

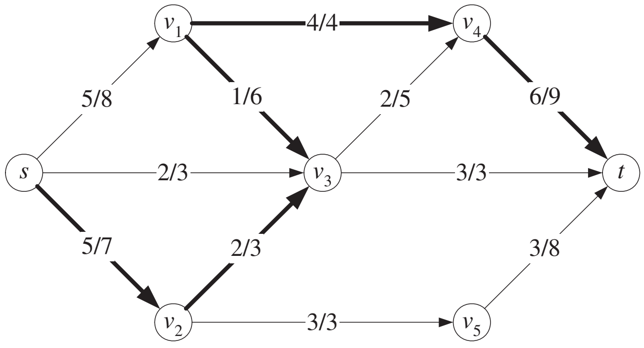
This value is the maximum amount of additional flow that we can possibly "push" down the path $\pi$ without violating a capacity constraint. An ***augmenting path*** for flow $f$ is a path $\pi$ from the source $s$ to the sink $t$ with nonzero residual capacity, that is, for each edge $e$ of $\pi$,

- $f(e) < c(e)$ if $e$ is a forward edge
- $f(e) > 0$ if $e$ is a backward edge.

We show in Figure 16.6 an example of an augmenting path.



(a)



(b)

**Figure 16.6:** Example of an augmenting path: (a) network $N$, flow $f$, and an augmenting path $\pi$ drawn with thick edges ($(v_1, v_3)$ is a backward edge); (b) flow $f'$ obtained from $f$ by pushing $\Delta_f(\pi) = 2$ units of flow from $s$ to $t$ along path $\pi$.

As shown by the following lemma, we can always add the residual capacity of an augmenting path to an existing flow and get another valid flow.

**Lemma 16.4:** *Let $\pi$ be an augmenting path for flow $f$ in network $N$. There exists a flow $f'$ for $N$ of value $|f'| = |f| + \Delta_f(\pi)$.*

**Proof:** We compute the flow $f'$ by modifying the flow of the edges of $\pi$:

$$f'(e) = \begin{cases} f(e) + \Delta_f(\pi) & \text{if } e \text{ is a forward edge} \\ f(e) - \Delta_f(\pi) & \text{if } e \text{ is a backward edge.} \end{cases}$$

Note that we subtract $\Delta_f(\pi)$ if $e$ is a backward edge, for we are subtracting flow on $e$ already taken by $f$ in this case. In any case, because $\Delta_f(\pi) \geq 0$ is the minimum residual capacity of any edge in $\pi$, we will violate no capacity constraint on a forward edge by adding $\Delta_f(\pi)$ nor will we go below zero flow on any backward edge by subtracting $\Delta_f(\pi)$. Thus, $f'$ is a valid flow for $N$, and the value of $f'$ is $|f| + \Delta_f(\pi)$. ■

By Lemma 16.4, the existence of an augmenting path $\pi$ for a flow $f$ implies that $f$ is not maximum. Also, given an augmenting path $\pi$, we can modify $f$ to increase its value by pushing $\Delta_f(\pi)$ units of flow from $s$ to $t$ along path $\pi$, as shown in the proof of Lemma 16.4.

What if there is no augmenting path for a flow $f$ in network $N$? In this case, we have that $f$ is a maximum flow, as stated by the following lemma.

**Lemma 16.5:** *If a network $N$ does not have an augmenting path with respect to a flow $f$, then $f$ is a maximum flow. Also, there is a cut $\chi$ of $N$ such that $|f| = c(\chi)$.*

**Proof:** Let $f$ be a flow for $N$, and suppose there is no augmenting path in $N$ with respect to $f$. We construct from $f$ a cut $\chi = (V_s, V_t)$ by placing in set $V_s$ all the vertices $v$, such that there is a path from the source $s$ to vertex $v$ consisting of edges of nonzero residual capacity. Such a path is called an augmenting path from $s$ to $v$. Set $V_t$ contains the remaining vertices of $N$. Since there is no augmenting path for flow $f$, the sink $t$ of $N$ is in $V_t$. Thus, $\chi = (V_s, V_t)$ satisfies the definition of a cut.

By the definition of $\chi$, each forward edge and backward edge of cut $\chi$ has zero residual capacity, that is,

$$f(e) = \begin{cases} c(e) & \text{if } e \text{ is a forward edge of } \chi \\ 0 & \text{if } e \text{ is a backward edge of } \chi. \end{cases}$$

Thus, the capacity of $\chi$ is equal to the value of $f$. That is,

$$|f| = c(\chi).$$

By Theorem 16.3, we have that $f$ is a maximum flow. ■

As a consequence of Theorem 16.3 and Lemma 16.5, we have the following:

**Theorem 16.6 (The Max-Flow, Min-Cut Theorem):** *The value of a maximum flow is equal to the capacity of a minimum cut.*

## 16.2 Maximum Flow Algorithms

In this section, we discuss two maximum flow algorithms in this section, starting with a classic algorithm, due to Ford and Fulkerson.

### 16.2.1 The Ford-Fulkerson Algorithm

The main idea of the ***Ford-Fulkerson algorithm*** is to incrementally increase the value of a flow in stages, where at each stage some amount of flow is pushed along an augmenting path from the source to the sink. Initially, the flow of each edge is equal to $0$. At each stage, an augmenting path $\pi$ is computed and an amount of flow equal to the residual capacity of $\pi$ is pushed along $\pi$, as in the proof of Lemma 16.4. The algorithm terminates when the current flow $f$ does not admit an augmenting path. Lemma 16.5 guarantees that $f$ is a maximum flow in this case.

We provide a pseudocode description of the Ford-Fulkerson solution to the problem of finding a maximum flow in Algorithm 16.7.

```
Algorithm MaxFlowFordFulkerson($N$):
  Input: Flow network $N = (G, c, s, t)$
  Output: A maximum flow $f$ for $N$
 for each edge $e \in N$ do
     $f(e) \leftarrow 0$
 stop ← false
 repeat
     traverse $G$ starting at $s$ to find an augmenting path for $f$
     if an augmenting path $\pi$ exists then
         // Compute the residual capacity $\Delta_f(\pi)$ of $\pi$
         $\Delta \leftarrow +\infty$
         for each edge $e \in \pi$ do
             if $\Delta_f(e) < \Delta$ then
                 $\Delta \leftarrow \Delta_f(e)$
         for each edge $e \in \pi$ do   // push $\Delta = \Delta_f(\pi)$ units along $\pi$
             if $e$ is a forward edge then
                 $f(e) \leftarrow f(e) + \Delta$
             else
                 $f(e) \leftarrow f(e) - \Delta$        // $e$ is a backward edge
     else
         stop ← true        // $f$ is a maximum flow
 until stop
```

**Algorithm 16.7:** The Ford-Fulkerson algorithm.

We visualize the Ford-Fulkerson algorithm in Figure 16.8.

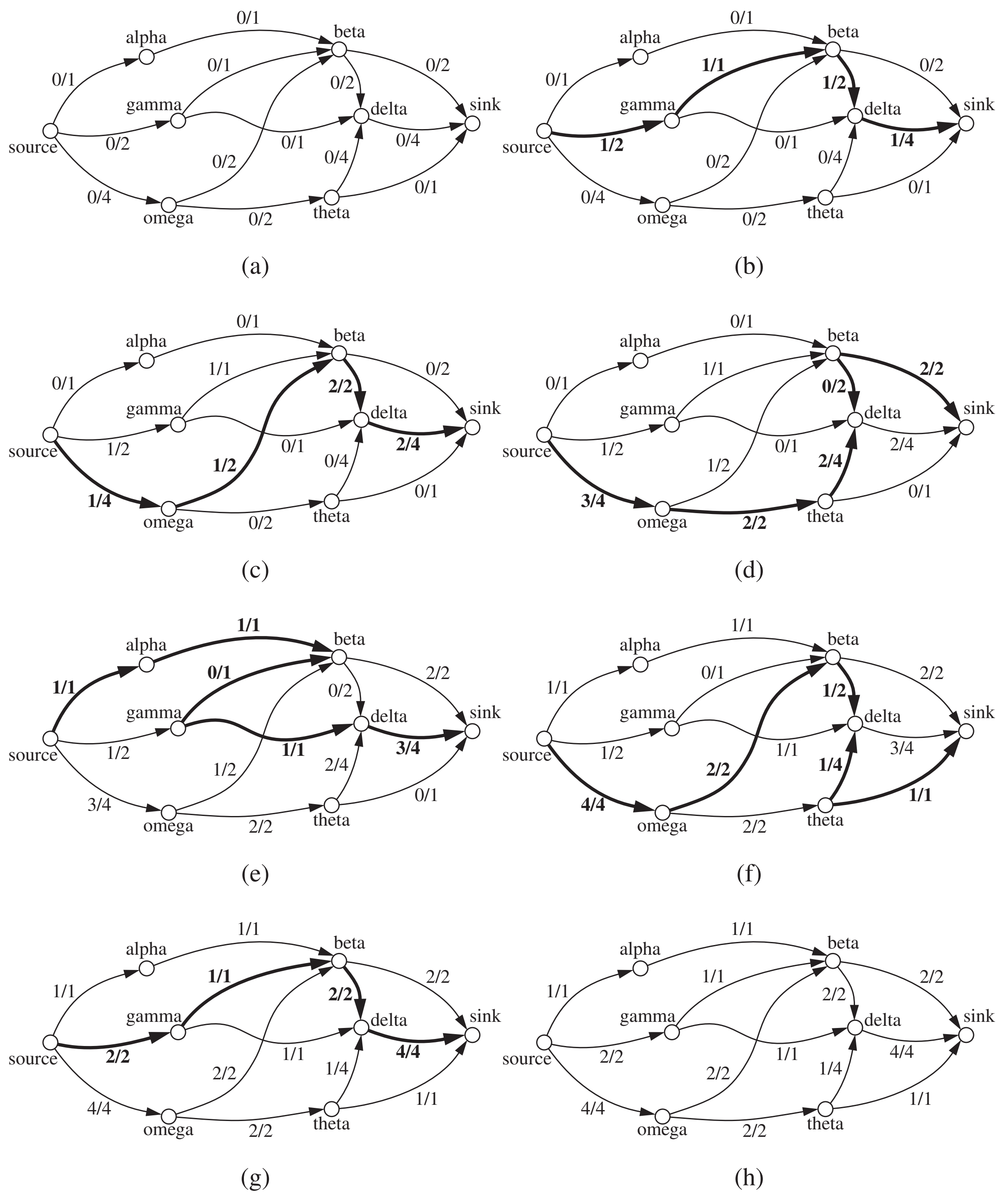


**Figure 16.8:** Example execution of the Ford-Fulkerson algorithm on the flow network of Figure 16.1. Augmenting paths are drawn with thick lines.

## Implementation Details

There are important implementation details for the Ford-Fulkerson algorithm that impact how we represent a flow and how we compute augmenting paths. Representing a flow is actually quite easy. We can label each edge of the network with an attribute representing the flow along that edge. To compute an augmenting path, we use a specialized traversal of the graph $G$ underlying the flow network. Such a traversal is a simple modification of either a DFS traversal (Section 13.2) or a BFS traversal (Section 13.3), where instead of considering all the edges incident on the current vertex $v$, we consider only the following edges:

- Outgoing edges of $v$ with flow less than the capacity
- Incoming edges of $v$ with nonzero flow.

Alternatively, the computation of an augmenting path with respect to the current flow $f$ can be reduced to a simple path-finding problem in a new directed graph $R_f$ derived from $G$. The vertices of $R_f$ are the same as the vertices of $G$. For each ordered pair of adjacent vertices $u$ and $v$ of $G$, we add a directed edge from $u$ to $v$ if $\Delta_f(u, v) > 0$. Graph $R_f$ is called the ***residual graph*** with respect to flow $f$. An augmenting path with respect to flow $f$ corresponds to a directed path from $s$ to $t$ in the residual graph $R_f$. This path can be computed by a DFS traversal of $R_f$ starting at the source.

## Analyzing the Ford-Fulkerson Algorithm

The analysis of the running time of the Ford-Fulkerson algorithm is a little tricky. This is because the algorithm does not specify the exact way to find augmenting paths and, as we shall see, the choice of augmenting path has a major impact on the algorithm's running time.

Let $n$ and $m$ be the number of vertices and edges of the flow network, respectively, and let $f^*$ be a maximum flow. Since the graph underlying the network is connected, we have that $n \leq m + 1$. Note that each time we find an augmenting path we increase the value of the flow by at least 1, since edge capacities and flows are integers. Thus, $|f^*|$, the value of a maximum flow, is an upper bound on the number of times the algorithm searches for an augmenting path. Also note that we can find an augmenting path by a simple graph traversal, such as a DFS or BFS traversal, which takes $O(m)$ time (see Theorems 13.13 and 13.15, and recall that $n \leq m + 1$). Thus, we can bound the running time of the Ford-Fulkerson algorithm as being at most $O(|f^*|m)$. As illustrated in Figure 16.9, this bound can actually be attained for some choices of augmenting paths. We conclude that the Ford-Fulkerson algorithm is a pseudo-polynomial-time algorithm (Section 12.6), since its running time depends on both the size of the input and also the value of a numeric parameter. Thus, the time bound of the Ford-Fulkerson algorithm can be quite slow if $|f^*|$ is large and augmenting paths are chosen poorly.

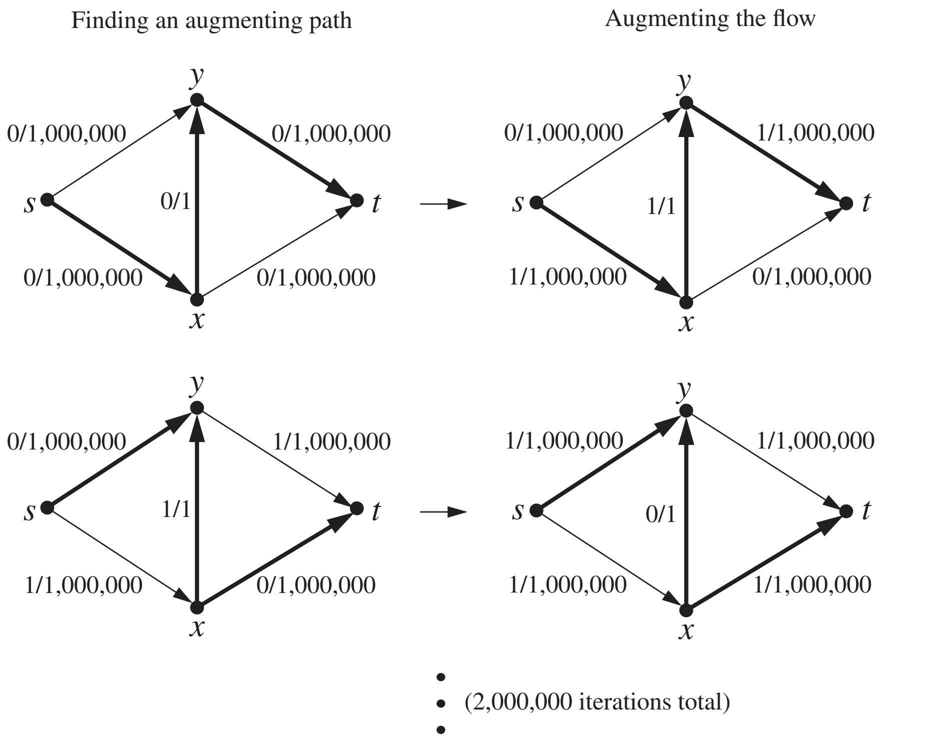


**Figure 16.9:** An example of a network for which the standard Ford-Fulkerson algorithm runs slowly. If the augmenting paths chosen by the algorithm alternate between $(s, x, y, t)$ and $(s, y, x, t)$, then the algorithm will make a total of $2,000,000$ iterations, even though two iterations would have sufficed.

### 16.2.2 The Edmonds-Karp Algorithm

The ***Edmonds-Karp algorithm*** is a variation of the Ford-Fulkerson algorithm. It uses a simple technique for finding good augmenting paths that results in a faster running time. This technique is based on the notion of being "more greedy" in our application of the greedy method to the maximum flow problem. Namely, at each iteration, we choose an augmenting path with the smallest number of edges, which can be easily done in $O(m)$ time by a modified BFS traversal. We will show that with these ***Edmonds-Karp augmentations***, the number of iterations is no more than $nm$, which implies an $O(nm^2)$ running time for the Edmonds-Karp algorithm.

We begin by introducing some notation. We call the ***length*** of a path $\pi$ the number of edges in $\pi$. Let $f$ be a flow for network $N$. Given a vertex $v$, we denote with $d_f(v)$ the minimum length of an augmenting path with respect to $f$ from the source $s$ to vertex $v$, and call this quantity the ***residual distance*** of $v$ with respect to flow $f$. The following discussion shows how residual distance of each vertex impacts the running time of the Edmonds-Karp algorithm.

## Performance of the Edmonds-Karp Algorithm

We begin our analysis by noting that residual distance is nondecreasing over a sequence of Edmonds-Karp augmentations.

**Lemma 16.7:** *Let $g$ be the flow obtained from flow $f$ with an augmentation along a path $\pi$ of minimum length. Then, for each vertex $v$,*

$$d_f(v) \le d_g(v).$$

**Proof:** Suppose there is a vertex violating the above inequality. Let $v$ be such a vertex with smallest residual distance with respect to $g$. That is,

$$d_f(v) > d_g(v) \tag{16.1}$$

and

$$d_g(v) \le d_g(u), \text{ for each } u \text{ such that } d_f(u) > d_g(u). \tag{16.2}$$

Consider an augmenting path $\gamma$ of minimum length from $s$ to $v$ with respect to flow $g$. Let $u$ be the vertex immediately preceding $v$ on $\gamma$, and let $e$ be the edge of $\gamma$ with endpoints $u$ and $v$ (see Figure 16.10). By the above definition, we have

$$\Delta_g(u,v) > 0. \tag{16.3}$$

Also, since $u$ immediately precedes $v$ in shortest path, $\gamma$, we have

$$d_g(v) = d_g(u) + 1. \tag{16.4}$$

Finally, by (16.2) and ( 16.4), we have

$$d_f(u) \le d_g(u). \tag{16.5}$$

We now show that $\Delta_f(u,v) = 0$. Indeed, if we had $\Delta_f(u,v) > 0$, we could go from $u$ to $v$ along an augmenting path with respect to flow $f$. This would imply

$$\begin{aligned} d_f(v) &\le d_f(u) + 1 \\ &\le d_g(u) + 1 && \text{by (16.5)} \\ &= d_g(v) && \text{by (16.4),} \end{aligned}$$

thus contradicting (16.1).

Since $\Delta_f(u,v) = 0$ and, by (16.3), $\Delta_g(u,v) > 0$, the augmenting path $\pi$, which produces $g$ from $f$, must traverse the edge $e$ from $v$ to $u$ (see Figure 16.10). Hence,

$$\begin{aligned} d_f(v) &= d_f(u) - 1 && \text{because } \pi \text{ is a shortest path} \\ &\le d_g(u) - 1 && \text{by (16.5)} \\ &\le d_g(v) - 2 && \text{by (16.4)} \\ &< d_g(v). \end{aligned}$$

Thus, we have obtained a contradiction with (16.1), which completes the proof. ■

Intuitively, Lemma 16.7 implies that each time we do an Edmonds-Karp augmentation, the residual distance from $s$ to any vertex $v$ can only increase or stay the same. This fact gives us the following.

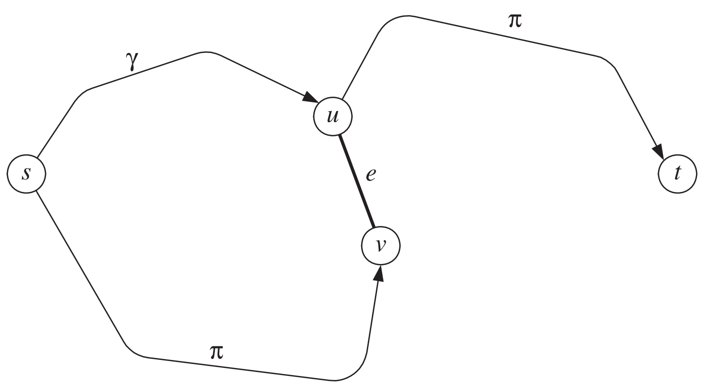


**Figure 16.10:** Illustration of the proof of Lemma 16.7.

**Lemma 16.8:** *When executing the Edmonds-Karp algorithm on a network with $n$ vertices and $m$ edges, the number of flow augmentations is no more than $nm$.*

**Proof:** Let $f_i$ be the flow in the network before the $i$th augmentation, and let $\pi_i$ be the path used in such augmentation. We say that an edge $e$ of $\pi_i$ is a ***bottleneck*** for $\pi_i$ if the residual capacity of $e$ is equal to the residual capacity of $\pi_i$. Clearly, every augmenting path used by the Edmonds-Karp algorithm has at least one bottleneck.

Consider a pair of vertices $u$ and $v$ joined by an edge $e$, and suppose that edge $e$ is a bottleneck for two augmenting paths $\pi_i$ and $\pi_k$, with $i < k$, that traverse $e$ from $u$ to $v$. The above assumptions imply each of the following:

- $\Delta_{f_i}(u, v) > 0$
- $\Delta_{f_{i+1}}(u, v) = 0$
- $\Delta_{f_k}(u, v) > 0$.

Thus, there must be an intermediate $j$th augmentation, with $i < j < k$ whose augmenting path $\pi_j$ traverses edge $e$ from $v$ to $u$. We therefore obtain

$$\begin{array}{rcll} d_{f_j}(u) & = & d_{f_j}(v) + 1 & \text{(because } \pi_j \text{ is a shortest path)} \\ & \geq & d_{f_i}(v) + 1 & \text{(by Lemma 16.7)} \\ & \geq & d_{f_i}(u) + 2 & \text{(because } \pi_i \text{ is a shortest path).} \end{array}$$

Since the residual distance of a vertex is always less than the number of vertices $n$, each edge can be a bottleneck at most $n$ times during the execution of the Edmonds-Karp algorithm ($n/2$ times for each of the two directions in which it can be traversed by an augmenting path). Hence, the overall number of augmentations is no more than $nm$. ■

Since a single flow augmentation can be done in $O(m)$ time using a modified BFS strategy, we can summarize the above discussion as follows.

**Theorem 16.9:** *Given a flow network with $n$ vertices and $m$ edges, the Edmonds-Karp algorithm computes a maximum flow in $O(nm^2)$ time.*

## 16.3 Maximum Bipartite Matching

A problem that arises in a number of important applications is the ***maximum bipartite matching*** problem. In this problem, we are given a connected undirected graph with the following properties:

- The vertices of $G$ are partitioned into two sets, $X$ and $Y$.
- Every edge of $G$ has one endpoint in $X$ and the other endpoint in $Y$.

Such a graph is called a ***bipartite graph***. A ***matching*** in $G$ is a set of edges that have no endpoints in common—such a set "pairs" up vertices in $X$ with vertices in $Y$ so that each vertex has at most one "partner" in the other set. The maximum bipartite matching problem is to find a matching with the greatest number of edges (over all matchings).

**Example 16.10:** *Let $G$ be a bipartite graph where the set $X$ represents a group of young men and the set $Y$ represents a group of young women, who are all together at a community dance. Let there be an edge joining $x$ in $X$ and $y$ in $Y$ if $x$ and $y$ are willing to dance with one another. A maximum matching in $G$ corresponds to a largest set of compatible pairs of men and women who can all be happily dancing at the same time.*

**Example 16.11:** *Let $G$ be a bipartite graph where the set $X$ represents a group of college courses and the set $Y$ represents a group of classrooms. Let there be an edge joining $x$ in $X$ and $y$ in $Y$ if, based on its enrollment and audiovisual needs, the course $x$ can be taught in classroom $y$. A maximum matching in $G$ corresponds to a largest set of college courses that can be taught simultaneously without conflicting.*

These two examples provide a small sample of the kinds of applications that the maximum bipartite matching problem can be used to solve. Fortunately, there is a simple way of solving the maximum bipartite matching problem.

### Reduction to the Maximum Flow Problem

Let $G$ be a bipartite graph whose vertices are partitioned into sets $X$ and $Y$. We create a flow network $H$ such that a maximum flow in $H$ can be immediately converted into a maximum matching in $G$:

- We begin by including all the vertices of $G$ in $H$, plus a new source vertex $s$ and a new sink vertex $t$.
- Next, we add every edge of $G$ to $H$, but direct each such edge so that it is oriented from the endpoint in $X$ to the endpoint in $Y$. In addition, we insert a directed edge from $s$ to each vertex in $X$, and a directed edge from each vertex in $Y$ to $t$. Finally, we assign to each edge of $H$ a capacity of 1.

Given a flow $f$ for $H$, we use $f$ to define a set $M$ of edges of $G$ using the rule that an edge $e$ is in $M$ whenever $f(e) = 1$. (See Figure 16.11.) We now show that the set $M$ is a matching. Since the capacities in $H$ are all 1, the flow through each edge of $H$ is either 0 or 1. Moreover, since each vertex $x$ in $X$ has exactly one incoming edge, the conservation rule implies that at most one outgoing edge of $x$ has nonzero flow. Similarly, since each vertex $y$ in $Y$ has exactly one outgoing edge, at most one incoming edge of $y$ has nonzero flow. Thus, each vertex in $X$ will be paired by $M$ with at most one vertex in $Y$, that is, set $M$ is a matching. Also, we can easily see that the size of $M$ is equal to $|f|$, the value of flow $f$.

A reverse transformation can also be defined. Namely, given a matching $M$ in graph $G$, we can use $M$ to define a flow $f$ for $H$ using the following rules:

- For each edge $e$ of $H$ that is also in $G$, $f(e) = 1$ if $e \in M$ and $f(e) = 0$ otherwise.
- For each edge $e$ of $H$ incident to $s$ or $t$, $f(e) = 1$ if $v$ is an endpoint of some edge of $M$ and $f(e) = 0$ otherwise, where $v$ denotes the other endpoint of $e$.

It is easy to verify that $f$ is a flow for $H$ and the value of $f$ is equal to the size of $M$.

Therefore, any maximum flow algorithm can be used to solve the maximum bipartite matching problem on a graph $G$ with $n$ vertices and $m$ edges. Namely,

1. We construct network $H$ from the bipartite graph $G$. This step takes $O(n + m)$ time. Network $H$ has $n + 2$ vertices and $n + m$ edges.
2. We compute a maximum flow for $H$ using the standard Ford-Fulkerson algorithm. Since the value of the maximum flow is equal to $|M|$, the size of the maximum matching, and $|M| \leq n/2$, this step takes $O(n(n + m))$ time, which is $O(nm)$ because $G$ is connected.

**Theorem 16.12:** *Let $G$ be a bipartite graph with $n$ vertices and $m$ edges. A maximum matching in $G$ can be computed in $O(nm)$ time.*

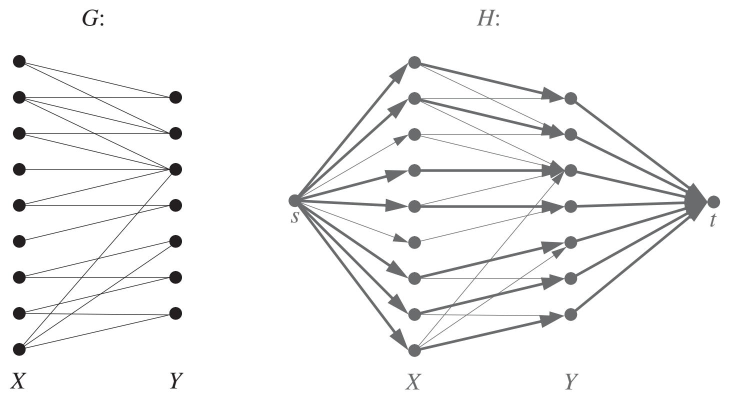


**Figure 16.11:** (a) A bipartite graph $G$. (b) Flow network $H$ derived from $G$ and a maximum flow in $H$; thick edges have unit flow and other edges have zero flow.

## 16.4 Baseball Elimination

Network flow has a lot of applications, with one of the more surprising being to a problem that arises in professional sports. Let $T$ be a set of teams in a sports league, which, for historical reasons, let us assume is baseball. At any point during the season, each team, $i$, in $T$, will have some number, $w_i$, of wins, and will have some number, $g_i$, of games left to play. The ***baseball elimination*** problem is to determine whether it is possible for team $i$ to finish the season in first place, given the games it has already won and the games it has left to play. Note that this depends on more than just the number of games left for team $i$, however; it also depends on the respective schedules of team $i$ and the other teams. So let $g_{i,j}$ denote the number of games remaining between team $i$ and team $j$, so that

$$g_i = \sum_{j \in T} g_{i,j}.$$

For example, see Table 16.12.

| Team | Wins | Games Left | Schedule $(g_{i,j})$ | | | |
|---|---|---|---|---|---|---|
| $i$ | $w_i$ | $g_i$ | LA | Oak | Sea | Tex |
| Los Angeles | 81 | 8 | - | 1 | 6 | 1 |
| Oakland | 77 | 4 | 1 | - | 0 | 3 |
| Seattle | 76 | 7 | 6 | 0 | - | 1 |
| Texas | 74 | 5 | 1 | 3 | 1 | - |

**Table 16.12:** A set of teams, their standings, and their remaining schedule. Clearly, Texas is eliminated from finishing in first place, since it can win at most 79 games. In addition, even though it is currently in second place, Oakland is also eliminated, because it can win at most 81 games, but in the remaining games between LA and Seattle, either LA wins at least 1 game and finishes with at least 82 wins or Seattle wins 6 games and finishes with at least 82 wins.

With all the different ways for a team, $k$, to be eliminated, it might at first seem like it is computationally infeasible to determine whether team $k$ is eliminated. Still, we can solve this problem by a reduction to a network flow problem. Let $T'$ denote the set of teams other than $k$, that is, $T' = T - \{k\}$. Also, let $L$ denote the set of games that are left to play among teams in $T'$, that is,

$$L = \{\{i, j\} : \ i, j \in T' \text{ and } g_{i,j} > 0\}.$$

Finally, let $W$ denote the largest number of wins that are possible for team $k$ given the current standings, that is, $W = w_k + g_k$.

If $W < w_i$, for some team $i$, then $k$ is eliminated directly by team $i$. So, let us assume that no single team eliminates team $k$. To consider how a combination of teams and game outcomes might eliminate team $k$, we create a graph, $G$, that has as its vertices a source, $s$, a sink, $t$, and the sets $T'$ and $L$. Then, let us include the following edges in $G$ (see Figure 16.13):

- For each game pair, $\{i, j\}$, in $L$, add an edge $(s,\ \{i, j\})$, and give it capacity $g_{i,j}$.
- For each game pair, $\{i, j\}$, in $L$, add edges $(\{i, j\},\ i)$ and $(\{i, j\},\ j)$, and give these edges capacity $+\infty$.
- For each team, $i$, add an edge $(i, t)$ and give it capacity $W - w_i$, which cannot be negative in this case, since we ruled out the case when $W < w_i$.

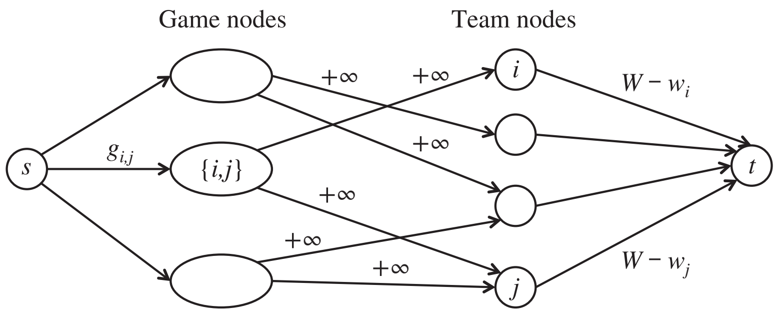


**Figure 16.13:** The network, $G$, to determine whether team $k$ is eliminated.

The intuition behind the construction for $G$ is that wins flow out from the source, $s$, are split at each game node, $\{i, j\}$, to allocate wins between each pair of teams, $i$ and $j$, and then are absorbed by the sink, $t$. The flow on each edge, $(\{i, j\},\ i)$, represents the number of games in which team $i$ beats $j$, and the flow on each edge, $(i, t)$, represents the number of remaining games that could be won by team $i$. Thus, maximizing the flow in $G$ is equivalent to testing if it is possible to allocate wins among all the remaining games not involving team $k$ so that no team goes above $W$ wins. So we compute a maximum flow for $G$.

Suppose that the value of this maximum flow is

$$g(T') = \sum_{\{i,j\} \subseteq T'} g_{i,j},$$

which is the total number of games to be played by teams in $T'$. This implies that it is possible to allocate wins to all the remaining games so that no team has its win count go above $W$, that is, team $k$ is not eliminated. If, on the other hand, the value of the maximum flow is strictly less than $g(T')$, then team $k$ is eliminated, since it is not possible to allocate wins to all the remaining games with every team having a win count of at most $W$. Thus, we have the following:

**Theorem 16.13:** *We can solve the baseball elimination problem for any team in a set of $n$ teams by solving a single maximum flow problem on a network with at most $O(n^2)$ vertices and edges.*

## 16.5 Minimum-Cost Flow

There is another variant of the maximum flow problem that applies in situations where there is a cost associated with sending a unit of flow through an edge. In this section, we extend the definition of a network by specifying a second nonnegative integer weight $w(e)$ for each edge $e$, representing the ***cost*** of edge $e$.

Given a flow $f$, we define the cost of $f$ as

$$w(f) = \sum_{e \in E} w(e) f(e),$$

where $E$ denotes the set of edges in the network. Flow $f$ is said to be a ***minimum-cost flow*** if $f$ has minimum cost among all flows of value $|f|$. The ***minimum-cost flow problem*** consists of finding a maximum flow that has the lowest cost over all maximum flows. A variation of the minimum-cost flow problem asks to find a minimum-cost flow with a given flow value. Given an augmenting path $\pi$ with respect to a flow $f$, we define the cost of $\pi$, denoted $w(\pi)$, as the sum of the costs of the forward edges of $\pi$ minus the sum of the costs of the backward edges of $\pi$.

An ***augmenting cycle*** with respect to flow $f$ is an augmenting path whose first and last vertices are the same. In more mathematical terms, it is a directed cycle $\gamma$ with vertices $v_0, v_1, \ldots, v_(k-1), v_k = v_0$, such that $\Delta_f(v_i, v_{i+1}) > 0$ for $i = 0, \ldots, k-1$ (see Figure 16.14). The definitions of residual capacity (given in Section 16.1.2) and cost (given above) also apply to an augmenting cycle. In addition, note that since it is a cycle, we can add the flow of an augmenting cycle to an existing flow without changing its flow value.

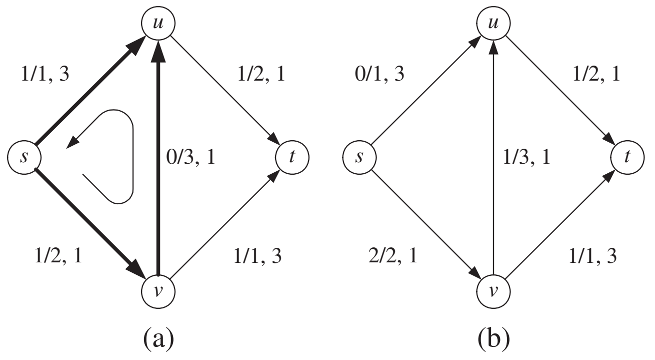


**Figure 16.14:** (a) Network with flow $f$, where each edge $e$ is labeled with $f(e)/c(e), w(e)$. We have $|f| = 2$ and $w(f) = 8$. Augmenting cycle $\gamma = (s, v, u, s)$, drawn with thick edges. The residual capacity of $\gamma$ is $\Delta_f(\gamma) = 1$. The cost of $\gamma$ is $w(\gamma) = -1$. (b) Flow $f'$ obtained from $f$ by pushing one unit of flow along cycle $\gamma$. We have $|f'| = |f|$ and $w(f') = w(f) + w(\gamma)\Delta_f(\gamma) = 8 + (-1) \cdot 1 = 7$.

### Adding the Flow from an Augmenting Cycle

The following lemma is analogous to Lemma 16.4, as it shows that a maximum flow can be changed into another maximum flow using an augmenting cycle.

**Lemma 16.14:** *Let $\gamma$ be an augmenting cycle for flow $f$ in network $N$. There exists a flow $f'$ for $N$ of value $|f'| = |f|$ and cost*

$$w(f') = w(f) + w(\gamma)\Delta_f(\gamma).$$

We leave the proof of Lemma 16.14 as an exercise (R-16.13).

### A Condition for Minimum-Cost Flows

Note that Lemma 16.14 implies that if a flow $f$ has an augmenting cycle of negative cost, then $f$ does not have minimum cost. The following theorem shows that the converse is also true, giving us a condition for testing when a flow is in fact a minimum-cost flow.

**Theorem 16.15:** *A flow $f$ has minimum cost among all flows of value $|f|$ if and only if there is no augmenting cycle of negative cost with respect to $f$.*

**Proof:** The "only-if" part follows immediately from Lemma 16.14. To prove the "if" part, suppose that flow $f$ does not have minimum cost, and let $g$ be a flow of value $f$ with minimum cost. Flow $g$ can be obtained from $f$ by a series of augmentations along augmenting cycles. Since the cost of $g$ is less than the cost of $f$, at least one of these cycles must have negative cost. ■

### An Algorithmic Approach for Finding Minimum-Cost Flows

Theorem 16.15 suggests an algorithm for the minimum-cost flow problem based on repeatedly augmenting flow along negative-cost cycles. We first find a maximum flow $f^*$ using the Ford-Fulkerson algorithm or the Edmonds-Karp algorithm. Next, we determine whether flow $f^*$ admits a negative-cost augmenting cycle. The Bellman-Ford algorithm (Section 14.3) can be used to find a negative cycle in time $O(nm)$. Let $w^*$ denote the total cost of the initial maximum flow $f^*$. After each execution of the Bellman-Ford algorithm, the cost of the flow decreases by at least one unit. Hence, starting from maximum flow $f^*$, we can compute a maximum flow of minimum cost in time $O(w^*nm)$. Therefore, we have the following:

**Theorem 16.16:** *Given an $n$-vertex flow network $N$ with costs associated with its $m$ edges, together with a maximum flow $f^*$, we can compute a maximum flow of minimum cost in $O(w^*nm)$ time, where $w^*$ is the total cost of $f^*$.*

We can do much better than this, however, by being more careful in how we compute augmenting cycles, as we show in the remainder of this section.

## Successive Shortest Paths

In this section, we present an alternative method for computing a minimum-cost flow. The idea is to start from an empty flow and build up to a maximum flow by a series of augmentations along minimum-cost paths. The following theorem provides the foundation of this approach.

**Theorem 16.17:** *Let $f$ be a minimum-cost flow, and let $f'$ be a the flow obtained by augmenting $f$ along an augmenting path $\pi$ of minimum cost. Flow $f'$ is a minimum-cost flow.*

**Proof:** The proof is illustrated in Figure 16.15.

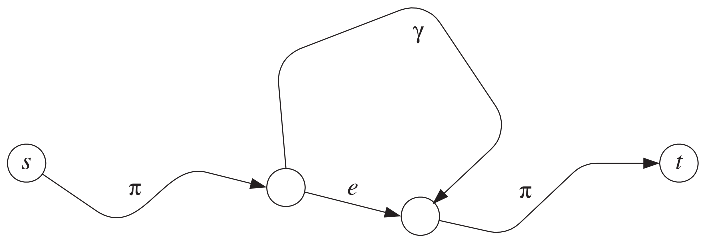


**Figure 16.15:** Illustrating the proof of Theorem 16.17.

Suppose, for the sake of a contradiction, that $f'$ does not have minimum cost. By Theorem 16.15, $f'$ has an augmenting cycle $\gamma$ of negative cost. Cycle $\gamma$ must have at least one edge $e$ in common with path $\pi$ and traverse $e$ in the direction opposite to that of $\pi$, since otherwise $\gamma$ would be an augmenting cycle of negative cost with respect to flow $f$, which is impossible, since $f$ has minimum cost. Consider the path $\hat{\pi}$ obtained from $\pi$ by replacing edge $e$ with $\gamma - e$. The path $\hat{\pi}$ is an augmenting path with respect to flow $f$. Also path $\hat{\pi}$ has cost

$$w(\hat{\pi}) = w(\pi) + w(\gamma) < w(\pi).$$

This contradicts the assumption that $\pi$ is an augmenting path of minimum cost with respect to flow $f$. ■

Starting from an initial null flow, we can compute a maximum flow of minimum cost by a repeated application of Theorem 16.17 (see Figure 16.16). Given the current flow $f$, we assign a weight to the edges of the residual graph $R_f$ as follows (recall the definition of residual graph from Section 16.2). For each edge $e$, directed from $u$ to $v$, of the original network, the edge of $R_f$ from $u$ to $v$, denoted $(u, v)$, has weight $w(u, v) = w(e)$, while the edge $(v, u)$ from $v$ to $u$ has weight $w(v, u) = -w(e)$. The computation of a shortest path in $R_f$ can be done by using the Bellman-Ford algorithm (see Section 14.3) since, by Theorem 16.15, $R_f$ does not have negative-cost cycles. Thus, we obtain a pseudo-polynomial-time algorithm (Section 12.6) that computes a maximum flow of minimum cost $f^*$ in time $O(|f^*|nm)$.

An example execution of the above algorithm is shown in Figure 16.16.

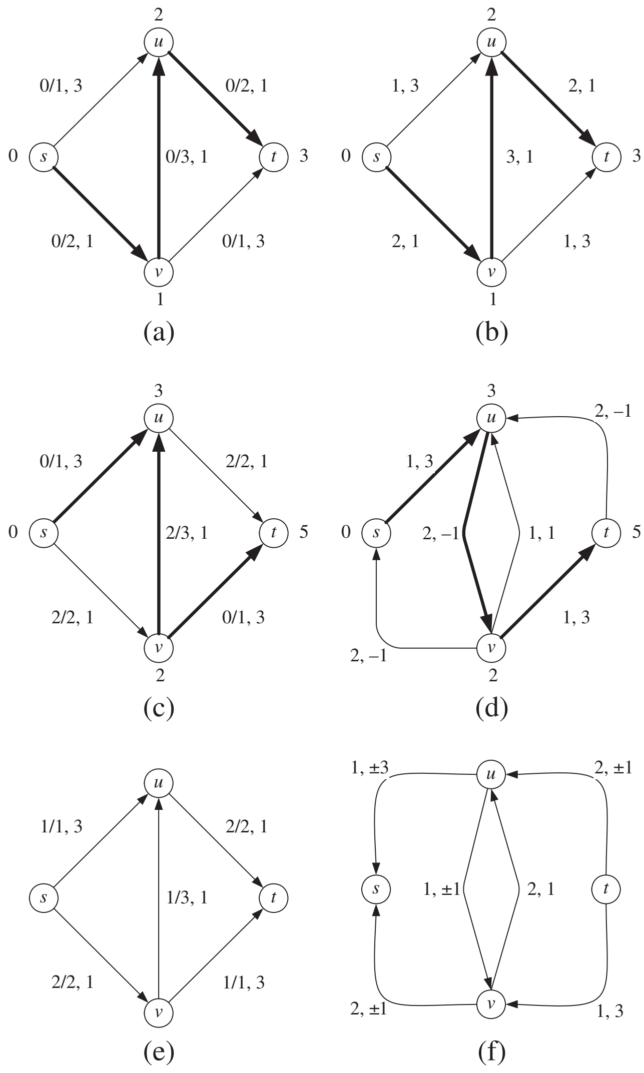


**Figure 16.16:** Example of computation of a minimum-cost flow by successive shortest-path augmentations. At each step, we show the network on the left and the residual network on the right. Vertices are labeled with their distance from the source. In the network, each edge $e$ is labeled with $f(e)/c(e), w(e)$. In the residual network, each edge is labeled with its residual capacity and cost (edges with zero residual capacity are omitted). Augmenting paths are drawn with thick lines. A minimum-cost flow is computed with two augmentations. In the first augmentation, two units of flow are pushed along path $(s, v, u, t)$. In the second augmentation, one unit of flow is pushed along path $(s, u, v, t)$.

### Modified Weights

We can reduce the time for the shortest-path computations by changing the weights in the residual graph $R_f$ so that they are all nonnegative. After the modification, we can use Dijkstra's algorithm, which runs in $O(m \log n)$ time, instead of the Bellman-Ford algorithm, which runs in $O(nm)$ time.

We describe now the modification of the edge weights. Let $f$ be the current minimum-cost flow. We denote with $d_f(v)$ the ***distance*** of vertex $v$ from the source $s$ in $R_f$, defined as the minimum weight of a path from $s$ to $v$ in $R_f$ (the cost of an augmenting path from the source $s$ to vertex $v$). Note that this definition of distance is different from the one used in Section 16.2.2 for the Edmonds-Karp algorithm.

Let $g$ be the flow obtained from $v$ by augmenting $f$ along a minimum-cost path. We define a new set of edge weights $w'$ for $R_g$, as follows (see Figure 16.17):

$$w'(u,v) = w(u,v) + d_f(u) - d_f(v).$$

**Lemma 16.18:** *For each edge $(u,v)$ of residual network $R_g$, we have*

$$w'(u,v) \geq 0.$$

*Also, a shortest path in $R_g$ with the modified edge weights $w'$ is also a shortest path with the original edge weights $w$.*

**Proof:** We distinguish two cases.

**Case 1:** The edge $(u,v)$ exists in $R_f$.
In this case, the distance $d_f(v)$ of $v$ from $s$ is no more than the distance $d_f(u)$ of $u$ from $s$ plus the weight $w(u,v)$ of edge $(u,v)$, that is,

$$d_f(v) \leq d_f(u) + w(u,v).$$

Thus, we have

$$w'(u,v) \geq 0.$$

**Case 2:** The edge $(u,v)$ does not exist in $R_f$.
In this case, $(v,u)$ must be an edge of the augmenting path used to obtained flow $g$ from flow $f$ and we have

$$d_f(u) = d_f(v) + w(v,u).$$

Since $w(v,u) = -w(u,v)$, we have

$$w'(u,v) = 0.$$

Given a path $\pi$ of $R_g$ from $s$ to $t$, the cost $w'(\pi)$ of $\pi$ with respect to the modified edge weights differs from the cost $c(\pi)$ of $\pi$ by a constant:

$$w'(\pi) = w(\pi) + d_f(s) - d_f(t) = w(\pi) - d_f(t).$$

Thus, a shortest path in $R_g$ with respect to the original weights is also a shortest path with respect to the modified weights. ■

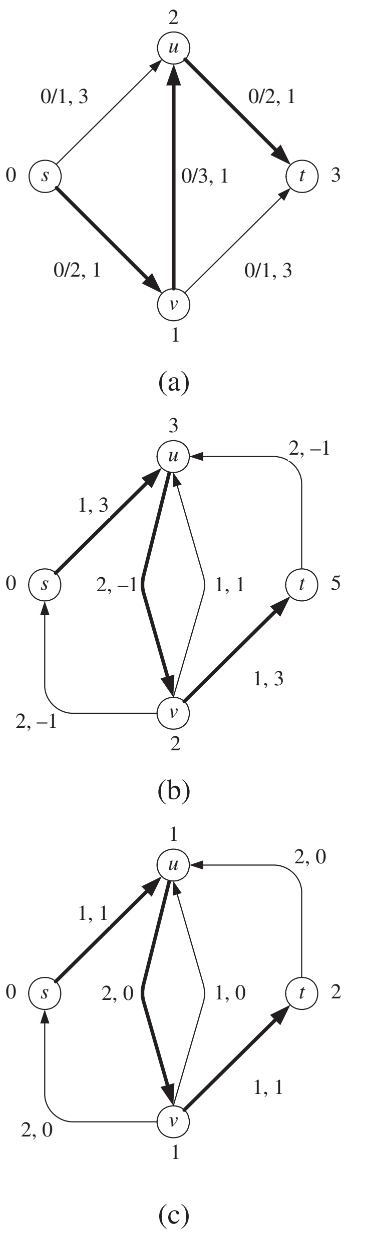


**Figure 16.17:** Modification of the edge costs in the computation of a minimum-cost flow by successive shortest-path augmentations. (a) Flow network $N_f$ with initial null flow $f$ and shortest augmenting path $\pi_1 = (s, v, u, t)$ with cost $w_1 = w(\pi_1) = 3$. Each vertex is labeled with its distance $d_f$ from the source. (Same as Figure 16.16.b.) (b) Residual network $R_g$ after augmenting flow $f$ by two units along path $\pi$ and shortest path $\pi_2 = (s, u, v, t)$ with cost $w(\pi_2) = 5$. (Same as Figure 16.16.d.) (c) Residual network $R_g$ with modified edge weights. Path $\pi_2$ is still a shortest path. However, its cost is decreased by $w_1$.

The complete algorithm for computing a minimum-cost flow using the successive shortest-path method is given in Algorithm 16.18 (MinCostFlow).

```
Algorithm MinCostFlow($N$):
   Input: Weighted flow network $N = (G, c, w, s, t)$
   Output: A maximum flow with minimum cost $f$ for $N$
  for each edge $e \in N$ do
      $f(e) \leftarrow 0$
  for each vertex $v \in N$ do
      $d(v) \leftarrow 0$
  stop ← false
  repeat
      compute the weighted residual network $R_f$
      for each edge $(u, v) \in R_f$ do
          $w'(u, v) \leftarrow w(u, v) + d(u) - d(v)$
      run Dijkstra's algorithm on $R_f$ using the weights $w'$
      for each vertex $v \in N$ do
          $d(v) \leftarrow$ distance of $v$ from $s$ in $R_f$
      if $d(t) < +\infty$ then
          // $\pi$ is an augmenting path with respect to $f$
          // Compute the residual capacity $\Delta_f(\pi)$ of $\pi$
          $\Delta \leftarrow +\infty$
          for each edge $e \in \pi$ do
              if $\Delta_f(e) < \Delta$ then
                  $\Delta \leftarrow \Delta_f(e)$
          // Push $\Delta = \Delta_f(\pi)$ units of flow along path $\pi$
          for each edge $e \in \pi$ do
              if $e$ is a forward edge then
                  $f(e) \leftarrow f(e) + \Delta$
              else
                  $f(e) \leftarrow f(e) - \Delta$        // $e$ is a backward edge
      else
          stop ← true        // $f$ is a maximum flow of minimum cost
  until stop
```

**Algorithm 16.18:** Successive shortest-path algorithm for computing a minimum-cost flow.

We summarize this section in the following theorem:

**Theorem 16.19:** *A minimum-cost maximum flow $f$ for a network with $n$ vertices and $m$ edges can be computed in $O(|f| m \log n)$ time.*

## 16.6 Exercises

### Reinforcement

**R-16.1** Show that for a flow $f$, the total flow out of the source is equal to the total flow into the sink, that is,

$$\sum_{e \in E^+(s)} f(e) = \sum_{e \in E^-(t)} f(e).$$

**R-16.2** Answer the following questions on the flow network $N$ and flow $f$ shown in Figure 16.6a:

- What are the forward and backward edges of augmenting path $\pi$?
- How many augmenting paths are there with respect to flow $f$? For each such path, list the sequence of vertices of the path and the residual capacity of the path.
- What is the value of a maximum flow in $N$?

**R-16.3** Construct a minimum cut for the network shown in Figure 16.4 using the method in the proof of Lemma 16.5.

**R-16.4** Illustrate the execution of the Ford-Fulkerson algorithm in the flow network of Figure 16.2.

**R-16.5** Draw a flow network with 9 vertices and 12 edges. Illustrate an execution of the Ford-Fulkerson algorithm on it.

**R-16.6** Find a minimum cut in the flow network of Figure 16.8a.

**R-16.7** Show that, given a maximum flow in a network with $m$ edges, a minimum cut of $N$ can be computed in $O(m)$ time.

**R-16.8** Find two maximum matchings for the bipartite graph of Figure 16.11a that are different from the maximum matching of Figure 16.11b.

**R-16.9** Let $G$ be a complete bipartite graph such that $|X| = |Y| = n$ and for each pair of vertices $x \in X$ and $y \in Y$, there is an edge joining $x$ and $y$. Show that $G$ has $n!$ distinct maximum matchings.

**R-16.10** Illustrate the execution of the Ford-Fulkerson algorithm in the flow network of Figure 16.11b.

**R-16.11** Illustrate the execution of the Edmonds-Karp algorithm in the flow network of Figure 16.8a.

**R-16.12** Illustrate the execution of the Edmonds-Karp algorithm in the flow network of Figure 16.2.

**R-16.13** Give a proof of Lemma 16.14.

**R-16.14** Illustrate the execution of the minimum-cost flow algorithm based on successive augmentations along negative-cost cycles for the flow network of Figure 16.16a.

**R-16.15** Illustrate the execution of the minimum-cost flow algorithm based on successive augmentations along minimum-cost paths for the flow network of Figure 16.2, where the cost of an edge $(u, v)$ is given by $|\text{deg}(u) - \text{deg}(v)|$.

**R-16.16** Is Algorithm 16.18 (MinCostFlow) a pseudo-polynomial-time algorithm?

## Creativity

**C-16.1** What is the worst-case running time of the Ford-Fulkerson algorithm if all edge capacities are bounded by a constant?

**C-16.2** Improve the bound of Lemma 16.8 by showing that there are at most $nm/4$ augmentations in the Edmonds-Karp algorithm.

***Hint:*** Use $d_f(u, t)$ in addition to $d_f(s, v)$.

**C-16.3** Let $N$ be a flow network with $n$ vertices and $m$ edges. Show how to compute an augmenting path with the largest residual capacity in $O((n + m)\log n)$ time.

**C-16.4** Show that the Ford-Fulkerson algorithm runs in time $O(m^2 \log n \log |f^*|)$ when, at each iteration, the augmenting path with the largest residual capacity is chosen.

**C-16.5** You want to increase the maximum flow of a network as much as possible, but you are only allowed to increase the capacity of one edge.

a. How do you find such an edge? (Give pseudocode.) You may assume the existence of algorithms to compute max flow and min cut. What's the running time of your algorithm?
b. Is it always possible to find such an edge? Justify your answer.

**C-16.6** Given a flow network $N$ and a maximum flow $f$ for $N$, suppose that the capacity of an edge $e$ of $N$ is decreased by one, and let $N'$ be the resulting network. Give an algorithm for computing a maximum flow in network $N'$ by modifying $f$.

**C-16.7** Give an algorithm that determines, in $O(n + m)$ time, whether a graph with $n$ vertices and $m$ edges is bipartite.

**C-16.8** Give an algorithm for computing a flow of maximum value subject to the following two additional constraints:

a. Each edge $e$ has a lower bound $\ell(e)$ on the flow through it.
b. There are multiple sources and sinks, and the value of the flow is computed as the total flow out of all the sources (equal to the total flow into all the sinks).

**C-16.9** Show that in a flow network with noninteger capacities, the Ford-Fulkerson algorithm may not terminate.

**C-16.10** In the context of the baseball elimination problem, one can show that if $w_i + g_i \leq w_k + g_k$ and team $k$ is eliminated, then team $i$ is also eliminated. Use this fact to show that among a set of $n$ teams, one can determine all the eliminated teams by solving $O(\log n)$ maximum flow problems.

**C-16.11** A ***vertex cover*** for a graph, $G$, is a set of vertices, $C$, such that every edge in $G$ is incident to one of the vertices in $C$. The problem of finding a smallest vertex cover is useful in network monitoring and other applications, but it is a difficult problem for general graphs. Show that the problem of finding a smallest vertex cover in a bipartite graph can be solved in polynomial time.

***Hint:*** Use the max-flow, min-cut theorem, and the reduction of Section 16.3, to prove that, for any bipartite graph, $G$, the number of vertices in a minimum vertex cover for $G$ equals the number of edges in a maximum matching in $G$.

## Applications

**A-16.1** In 2006, the city of Beijing, China, instituted a policy that limits residents to own at most one dog per household. Imagine you are running an online pet adoption website for the city. Your website contains pictures of adorable puppies that are available for adoption, and it allows for dogless Beijing residents to click on as many puppies as they like, with the understanding that they can adopt at most one. Suppose now that you have collected the puppy preferences from among $n$ Beijing residents for your $m$ puppies. Describe an efficient algorithm for assigning puppies to residents that provides for the maximum number of puppy adoptions possible while satisfying the constraints that each resident will only adopt a puppy that he or she likes and that no resident can adopt more than one puppy.

**A-16.2** The city of Irvine, California, allows for residents to own a maximum of three dogs per household without a breeder's license. Imagine you are running an online pet adoption website for the city, as in the previous exercise, but now for $n$ Irvine residents and $m$ puppies. Describe an efficient algorithm for assigning puppies to residents that provides for the maximum number of puppy adoptions possible while satisfying the constraints that each resident will only adopt puppies that he or she likes and that no resident can adopt more than three puppies.

**A-16.3** Consider the previous exercise, but suppose the city of Irvine, California, changed its dog-owning ordinance so that it still allows for residents to own a maximum of three dogs per household, but now restricts each resident to own at most one dog of any given breed, such as poodle, terrier, or golden retriever. Describe an efficient algorithm for assigning puppies to residents that provides for the maximum number of puppy adoptions possible while satisfying the constraints that each resident will only adopt puppies that he or she likes, that no resident can adopt more than three puppies, and that no resident will adopt more than one dog of any given breed.

**A-16.4** Imagine that you are working on creating a flow for a set of packets in a media stream, as described in the introduction to this chapter. So you are given a network, $G$, with a source, $s$, and sink, $t$, together with bandwidth constraints on each edge, which indicate the maximum speed that the communication link represented by that edge can support. As mentioned before, your goal is to produce a maximum flow from $s$ to $t$, respecting the bandwidth constraints on the edges. Suppose now, however, that you also have a bandwidth constraint on each router in the network, which specifies the maximum amount of information, in bits per second, that can pass through that node. Describe an efficient algorithm for finding a maximum flow in the network, $G$, that satisfies the bandwidth capacity constraints on the edges as well as the vertices. What is the running time of your algorithm?

**A-16.5** Suppose, as an interview question, you are told that you have a goat and a wolf that need to go from a node, $s$, to a node, $t$, in a directed acyclic graph, $G$. To avoid the wolf eating the goat, their paths must never share an edge. Describe a polynomial-time algorithm for finding two edge-disjoint paths in $G$, if such paths exist, to provide a way for the goat and the wolf to go from $s$ to $t$ without risk to the goat.

**A-16.6** Suppose a friend of yours has created a simulation game based on J.R.R. Tolkien's epic *The Lord of the Rings*. The game environment is Middle Earth, which is populated by various noble creatures, including hobbits, humans, dwarves, and elves. Unfortunately, these noble creatures are under attack and need to get to safe havens, known as "strongholds." Some strongholds are larger than others, of course, and each stronghold, $s$, can only hold some number, $N_s$, of these creatures. Initially, let us assume each stronghold is empty, and the noble creatures are living in various regions, with each region, $r$, containing some number, $N_r$, of noble creatures. Moreover, we know, for each region, $r$, the set of strongholds, $S_r$, that can be reached from $r$ in at most three days' travel. Your job is to figure out how to move the maximum number of noble creatures possible from the regions where they currently live to the various strongholds in three days' time while not overcrowding any stronghold. Describe and analyze an efficient algorithm to solve this game.

**A-16.7** A limousine company must process pickup requests every day, for taking customers from their various homes to the local airport. Suppose this company receives pickup requests from $n$ locations and there are $n$ limos available, where the distance of limo $i$ to location $j$ is given by a number, $d_{ij}$. Describe an efficient algorithm for computing a dispatchment of the $n$ limos to the $n$ pickup locations that minimizes the total distance traveled by all the limos.

## Chapter Notes

Ford and Fulkerson's network flow algorithm (16.2) is described in their book [74]. Edmonds and Karp [64] describe two methods for computing augmenting paths that cause the Ford-Fulkerson algorithm to run faster: shortest augmenting path (Section 16.2.2) and augmenting paths with maximum residual capacity (Exercise C-16.4). The minimum-cost flow algorithm based on successive augmentations along minimum-cost paths (Section 16.5) is also due to Edmonds and Karp [64].

The reader interested in further study of graph algorithms and flow networks is referred to the books by Ahuja, Magnanti, and Orlin [10], Even [68], Gibbons [81], Mehlhorn [158], and Tarjan [207], and the book chapter by van Leeuwen [210]. Schwartz [187] was the first to show that the baseball elimination problem could be reduced to a maximum flow problem. Our formulation of the baseball elimination problem follows that of Wayne [215]. For applications of network flow to social networks, see the book by Easley and Kleinberg [60].