26 Maximum Flow

Just as we can model a road map as a directed graph in order to find the shortest path from one point to another, we can also interpret a directed graph as a "flow network" and use it to answer questions about material flows. Imagine a material coursing through a system from a source, where the material is produced, to a sink, where it is consumed. The source produces the material at some steady rate, and the sink consumes the material at the same rate. The "flow" of the material at any point in the system is intuitively the rate at which the material moves. Flow networks can be used to model liquids flowing through pipes, parts through assembly lines, current through electrical networks, information through communication networks, and so forth.

Each directed edge in a flow network can be thought of as a conduit for the material. Each conduit has a stated capacity, given as a maximum rate at which the material can flow through the conduit, such as 200 gallons of liquid per hour through a pipe or 20 amperes of electrical current through a wire. Vertices are conduit junctions, and other than the source and sink, material flows through the vertices without collecting in them. In other words, the rate at which material enters a vertex must equal the rate at which it leaves the vertex. We call this property "flow conservation," and it is equivalent to Kirchhoff's Current Law when the material is electrical current.

In the maximum-flow problem, we wish to compute the greatest rate at which material can be shipped from the source to the sink without violating any capacity constraints. It is one of the simplest problems concerning flow networks and, as we shall see in this chapter, this problem can be solved by efficient algorithms. Moreover, the basic techniques used in maximum-flow algorithms can be adapted to solve other network-flow problems.

This chapter presents two general methods for solving the maximum-flow problem. Section 26.1 formalizes the notions of flow networks and flows, formally defining the maximum-flow problem. Section 26.2 describes the classical method of Ford and Fulkerson for finding maximum flows. An application of this method, finding a maximum matching in an undirected bipartite graph, is given in Section 26.3. Section 26.4 presents the push-relabel method, which underlies many of the fastest algorithms for network-flow problems. Section 26.5 covers the "relabel-to-front" algorithm, a particular implementation of the push-relabel method that runs in time $O(V^3)$. Although this algorithm is not the fastest algorithm known, it illustrates some of the techniques used in the asymptotically fastest algorithms, and it is reasonably efficient in practice.

26.1 Flow networks

In this section, we give a graph-theoretic definition of flow networks, discuss their properties, and define the maximum-flow problem precisely. We also introduce some helpful notation.

Flow networks and flows

A ***flow network*** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v) \geq 0$. If $(u, v) \notin E$, we assume that $c(u, v) = 0$. We distinguish two vertices in a flow network: a ***source*** s and a ***sink*** t. For convenience, we assume that every vertex lies on some path from the source to the sink. That is, for every vertex $v \in V$, there is a path $s \leadsto v \leadsto t$. The graph is therefore connected, and $|E| \geq |V| - 1$. Figure 26.1 shows an example of a flow network.

We are now ready to define flows more formally. Let $G = (V, E)$ be a flow network with a capacity function c. Let s be the source of the network, and let t be the sink. A ***flow*** in G is a real-valued function $f : V \times V \to \mathbf{R}$ that satisfies the following three properties:

Capacity constraint: For all $u, v \in V$, we require $f(u, v) \leq c(u, v)$.

Skew symmetry: For all $u, v \in V$, we require $f(u, v) = -f(v, u)$.

Flow conservation: For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(u, v) = 0 .$$

The quantity $f(u, v)$, which can be positive, zero, or negative, is called the ***flow*** from vertex u to vertex v. The ***value*** of a flow f is defined as

$$|f| = \sum_{v \in V} f(s, v) , \tag{26.1}$$

that is, the total flow out of the source. (Here, the $|\cdot|$ notation denotes flow value, not absolute value or cardinality.) In the ***maximum-flow problem***, we are given a

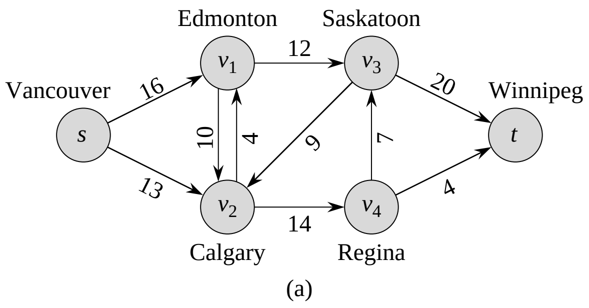


Figure 26.1 **(a)** A flow network $G = (V, E)$ for the Lucky Puck Company's trucking problem. The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t. Pucks are shipped through intermediate cities, but only $c(u, v)$ crates per day can go from city u to city v. Each edge is labeled with its capacity. **(b)** A flow f in G with value $|f| = 19$. Only positive flows are shown. If $f(u, v) > 0$, edge (u, v) is labeled by $f(u, v)/c(u, v)$. (The slash notation is used merely to separate the flow and capacity; it does not indicate division.) If $f(u, v) \le 0$, edge (u, v) is labeled only by its capacity.

flow network G with source s and sink t, and we wish to find a flow of maximum value.

Before seeing an example of a network-flow problem, let us briefly explore the three flow properties. The capacity constraint simply says that the flow from one vertex to another must not exceed the given capacity. Skew symmetry is a notational convenience that says that the flow from a vertex u to a vertex v is the negative of the flow in the reverse direction. The flow-conservation property says that the total flow out of a vertex other than the source or sink is 0. By skew symmetry, we can rewrite the flow-conservation property as

$$\sum_{u \in V} f(u, v) = 0$$

for all $v \in V - \{s, t\}$. That is, the total flow into a vertex is 0.

When neither (u, v) nor (v, u) is in E, there can be no flow between u and v, and $f(u, v) = f(v, u) = 0$. (Exercise 26.1-1 asks you to prove this property formally.)

Our last observation concerning the flow properties deals with flows that are positive. The ***total positive flow*** entering a vertex v is defined by

$$\sum_{\substack{u \in V \\ f(u,v) > 0}} f(u, v) \; . \tag{26.2}$$

The total positive flow leaving a vertex is defined symmetrically. We define the ***total net flow*** at a vertex to be the total positive flow leaving a vertex minus the total positive flow entering a vertex. One interpretation of the flow-conservation property is that the total positive flow entering a vertex other than the source or sink must equal the total positive flow leaving that vertex. This property, that the total net flow at a vertex must equal 0, is often informally referred to as “flow in equals flow out.”

An example of flow

A flow network can model the trucking problem shown in Figure 26.1(a). The Lucky Puck Company has a factory (source s) in Vancouver that manufactures hockey pucks, and it has a warehouse (sink t) in Winnipeg that stocks them. Lucky Puck leases space on trucks from another firm to ship the pucks from the factory to the warehouse. Because the trucks travel over specified routes (edges) between cities (vertices) and have a limited capacity, Lucky Puck can ship at most $c(u, v)$ crates per day between each pair of cities u and v in Figure 26.1(a). Lucky Puck has no control over these routes and capacities and so cannot alter the flow network shown in Figure 26.1(a). Their goal is to determine the largest number p of crates per day that can be shipped and then to produce this amount, since there is no point in producing more pucks than they can ship to their warehouse. Lucky Puck is not concerned with how long it takes for a given puck to get from the factory to the warehouse; they care only that p crates per day leave the factory and p crates per day arrive at the warehouse.

On the surface, it seems appropriate to model the “flow” of shipments with a flow in this network because the number of crates shipped per day from one city to another is subject to a capacity constraint. Additionally, flow conservation must be obeyed, for in a steady state, the rate at which pucks enter an intermediate city must equal the rate at which they leave. Otherwise, crates would accumulate at intermediate cities.

There is one subtle difference between shipments and flows, however. Lucky Puck may ship pucks from Edmonton to Calgary, and they may also ship pucks from Calgary to Edmonton. Suppose that they ship 8 crates per day from Edmonton (v_1 in Figure 26.1) to Calgary (v_2) and 3 crates per day from Calgary to Edmonton. It may seem natural to represent these shipments directly by flows, but we cannot. The skew-symmetry constraint requires that $f(v_1, v_2) = -f(v_2, v_1)$, but this is clearly not the case if we consider $f(v_1, v_2) = 8$ and $f(v_2, v_1) = 3$.

Lucky Puck may realize that it is pointless to ship 8 crates per day from Edmonton to Calgary and 3 crates from Calgary to Edmonton, when they could achieve the same net effect by shipping 5 crates from Edmonton to Calgary and 0 crates from Calgary to Edmonton (and presumably use fewer resources in the process). We represent this latter scenario with a flow: we have $f(v_1, v_2) = 5$ and $f(v_2, v_1) = -5$. In effect, 3 of the 8 crates per day from v_1 to v_2 are ***canceled*** by 3 crates per day from v_2 to v_1.

In general, cancellation allows us to represent the shipments between two cities by a flow that is positive along at most one of the two edges between the corresponding vertices. That is, any situation in which pucks are shipped in both directions between two cities can be transformed using cancellation into an equivalent situation in which pucks are shipped in one direction only: the direction of positive flow.

Given a flow f that arose from, say, physical shipments, we cannot reconstruct the exact shipments. If we know that $f(u, v) = 5$, this flow may be because 5 units were shipped from u to v, or it may be because 8 units were shipped from u to v and 3 units were shipped from v to u. Typically, we shall not care how the actual physical shipments are set up; for any pair of vertices, we care only about the net amount that travels between them. If we do care about the underlying shipments, then we should be using a different model, one that retains information about shipments in both directions.

Cancellation will arise implicitly in the algorithms in this chapter. Suppose that edge (u, v) has a flow value of $f(u, v)$. In the course of an algorithm, we may increase the flow on edge (v, u) by some amount d. Mathematically, this operation must decrease $f(u, v)$ by d and, conceptually, we can think of these d units as canceling d units of flow that are already on edge (u, v).

Networks with multiple sources and sinks

A maximum-flow problem may have several sources and sinks, rather than just one of each. The Lucky Puck Company, for example, might actually have a set of m factories $\{s_1, s_2, \ldots, s_m\}$ and a set of n warehouses $\{t_1, t_2, \ldots, t_n\}$, as shown in Figure 26.2(a). Fortunately, this problem is no harder than ordinary maximum flow.

We can reduce the problem of determining a maximum flow in a network with multiple sources and multiple sinks to an ordinary maximum-flow problem. Figure 26.2(b) shows how the network from (a) can be converted to an ordinary flow network with only a single source and a single sink. We add a ***supersource*** s and add a directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2, \ldots, m$. We also create a new ***supersink*** t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, \ldots, n$. Intuitively, any flow in the network in (a) corresponds to a flow in the network in (b), and vice versa. The single source s simply provides as much flow as desired for the multiple sources s_i, and the single sink t likewise consumes as much flow as desired for the multiple sinks t_i. Exercise 26.1-3 asks you to prove formally that the two problems are equivalent.

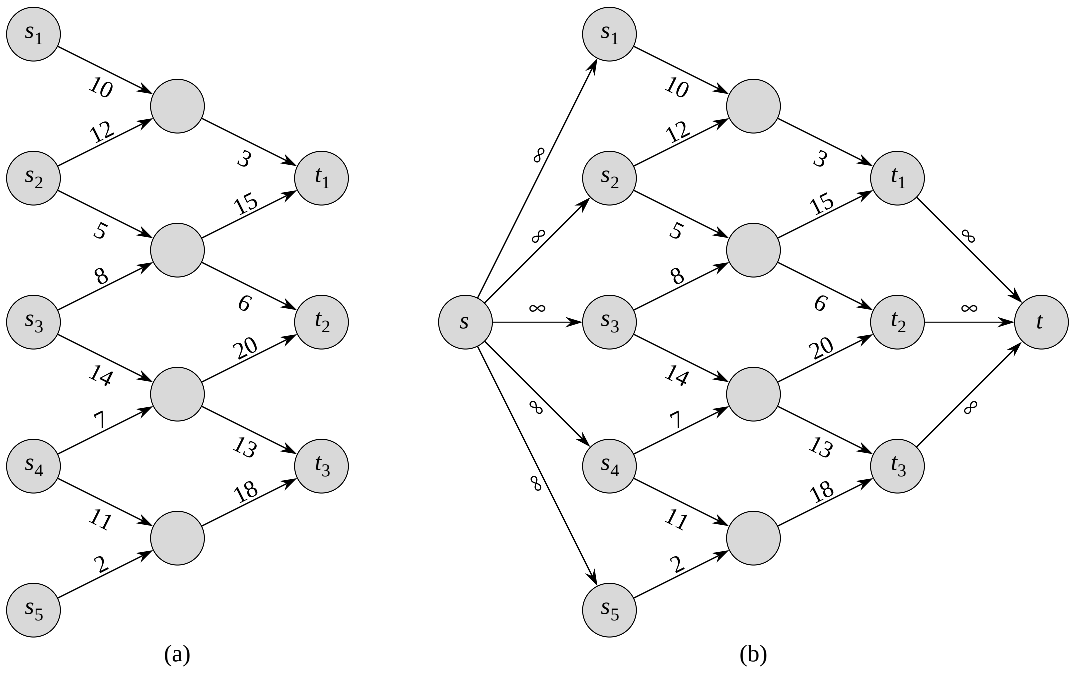


Figure 26.2 Converting a multiple-source, multiple-sink maximum-flow problem into a problem with a single source and a single sink. **(a)** A flow network with five sources $S = \{s_1, s_2, s_3, s_4, s_5\}$ and three sinks $T = \{t_1, t_2, t_3\}$. **(b)** An equivalent single-source, single-sink flow network. We add a supersource s and an edge with infinite capacity from s to each of the multiple sources. We also add a supersink t and an edge with infinite capacity from each of the multiple sinks to t.

Working with flows

We shall be dealing with several functions (like f) that take as arguments two vertices in a flow network. In this chapter, we shall use an ***implicit summation notation*** in which either argument, or both, may be a *set* of vertices, with the interpretation that the value denoted is the sum of all possible ways of replacing the arguments with their members. For example, if X and Y are sets of vertices, then

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y) .$$

Thus, the flow-conservation constraint can be expressed as the condition that $f(u, V) = 0$ for all $u \in V - \{s, t\}$. Also, for convenience, we shall typically omit set braces when they would otherwise be used in the implicit summation notation. For example, in the equation $f(s, V - s) = f(s, V)$, the term $V - s$ means the set $V - \{s\}$.

The implicit summation notation often simplifies equations involving flows. The following lemma, whose proof is left as Exercise 26.1-4, captures several of the most commonly occurring identities that involve flows and the implicit summation notation.

Lemma 26.1
Let $G = (V, E)$ be a flow network, and let f be a flow in G. Then the following equalities hold:

1. For all $X \subseteq V$, we have $f(X, X) = 0$.
2. For all $X, Y \subseteq V$, we have $f(X, Y) = -f(Y, X)$.
3. For all $X, Y, Z \subseteq V$ with $X \cap Y = \emptyset$, we have the sums $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ and $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$. ■

As an example of working with the implicit summation notation, we can prove that the value of a flow is the total flow into the sink; that is,

$$|f| = f(V, t) . \tag{26.3}$$

Intuitively, we expect this property to hold. By flow conservation, all vertices other than the source and sink have equal amounts of total positive flow entering and leaving. The source has, by definition, a total net flow that is greater than 0; that is, more positive flow leaves the source than enters it. Symmetrically, the sink is the only vertex that can have a total net flow that is less than 0; that is, more positive flow enters the sink than leaves it. Our formal proof goes as follows:

$$\begin{aligned}
|f| &= f(s, V) && \text{(by definition)} \\
&= f(V, V) - f(V - s, V) && \text{(by Lemma 26.1, part (3))} \\
&= -f(V - s, V) && \text{(by Lemma 26.1, part (1))} \\
&= f(V, V - s) && \text{(by Lemma 26.1, part (2))} \\
&= f(V, t) + f(V, V - s - t) && \text{(by Lemma 26.1, part (3))} \\
&= f(V, t) && \text{(by flow conservation)} .
\end{aligned}$$

Later in this chapter, we shall generalize this result (Lemma 26.5).

Exercises

26.1-1
Using the definition of a flow, prove that if $(u, v) \notin E$ and $(v, u) \notin E$ then $f(u, v) = f(v, u) = 0$.

26.1-2
Prove that for any vertex v other than the source or sink, the total positive flow entering v must equal the total positive flow leaving v.

26.1-3
Extend the flow properties and definitions to the multiple-source, multiple-sink problem. Show that any flow in a multiple-source, multiple-sink flow network corresponds to a flow of identical value in the single-source, single-sink network obtained by adding a supersource and a supersink, and vice versa.

26.1-4
Prove Lemma 26.1. You should not need to use flow conservation in your proof.

26.1-5
For the flow network $G = (V, E)$ and flow f shown in Figure 26.1(b), find a pair of subsets $X, Y \subseteq V$ for which $f(X, Y) = -f(V - X, Y)$. Then, find a pair of subsets $X, Y \subseteq V$ for which $f(X, Y) \neq -f(V - X, Y)$.

26.1-6
Given a flow network $G = (V, E)$, let f_1 and f_2 be functions from $V \times V$ to $\mathbf{R}$. The ***flow sum*** $f_1 + f_2$ is the function from $V \times V$ to $\mathbf{R}$ defined by

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \tag{26.4}$$

for all $u, v \in V$. If f_1 and f_2 are flows in G, which of the three flow properties must the flow sum $f_1 + f_2$ satisfy, and which might it violate?

26.1-7
Let f be a flow in a network, and let α be a real number. The ***scalar flow product***, denoted αf, is a function from $V \times V$ to $\mathbf{R}$ defined by

$$(\alpha f)(u, v) = \alpha \cdot f(u, v) .$$

Prove that the flows in a network form a ***convex set***. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha) f_2$ for all α in the range $0 \leq \alpha \leq 1$.

26.1-8
State the maximum-flow problem as a linear-programming problem.

26.1-9
Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining if both his children can go to the same school as a maximum-flow problem.

26.2 The Ford-Fulkerson method

This section presents the Ford-Fulkerson method for solving the maximum-flow problem. We call it a "method" rather than an "algorithm" because it encompasses several implementations with differing running times. The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems: residual networks, augmenting paths, and cuts. These ideas are essential to the important max-flow min-cut theorem (Theorem 26.7), which characterizes the value of a maximum flow in terms of cuts of the flow network. We end this section by presenting one specific implementation of the Ford-Fulkerson method and analyzing its running time.

The Ford-Fulkerson method is iterative. We start with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value by finding an "augmenting path," which we can think of simply as a path from the source s to the sink t along which we can send more flow, and then augmenting the flow along this path. We repeat this process until no augmenting path can be found. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

```
FORD-FULKERSON-METHOD($G, s, t$)
1  initialize flow $f$ to 0
2  while there exists an augmenting path $p$
3      do augment flow $f$ along $p$
4  return $f$
```

Residual networks

Intuitively, given a flow network and a flow, the residual network consists of edges that can admit more flow. More formally, suppose that we have a flow network $G = (V, E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u, v \in V$. The amount of *additional* flow we can push from u to v before exceeding the capacity $c(u, v)$ is the ***residual capacity*** of (u, v), given by

$$c_f(u, v) = c(u, v) - f(u, v) . \tag{26.5}$$

For example, if $c(u, v) = 16$ and $f(u, v) = 11$, then we can increase $f(u, v)$ by $c_f(u, v) = 5$ units before we exceed the capacity constraint on edge (u, v). When the flow $f(u, v)$ is negative, the residual capacity $c_f(u, v)$ is greater than the capacity $c(u, v)$. For example, if $c(u, v) = 16$ and $f(u, v) = -4$, then the residual capacity $c_f(u, v)$ is 20. We can interpret this situation as follows. There is a flow of 4 units from v to u, which we can cancel by pushing a flow of 4 units

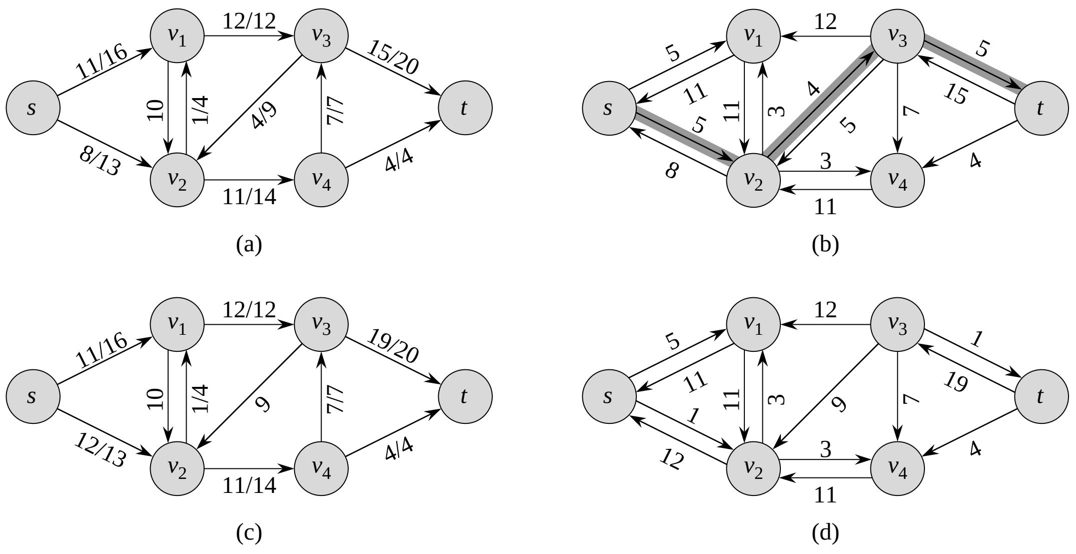


Figure 26.3 **(a)** The flow network G and flow f of Figure 26.1(b). **(b)** The residual network G_f with augmenting path p shaded; its residual capacity is $c_f(p) = c_f(v_2, v_3) = 4$. **(c)** The flow in G that results from augmenting along path p by its residual capacity 4. **(d)** The residual network induced by the flow in (c).

from u to v. We can then push another 16 units from u to v before violating the capacity constraint on edge (u, v). We have thus pushed an additional 20 units of flow, starting with a flow $f(u, v) = -4$, before reaching the capacity constraint.

Given a flow network $G = (V, E)$ and a flow f, the ***residual network*** of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\} \ .$$

That is, as promised above, each edge of the residual network, or ***residual edge***, can admit a flow that is greater than 0. Figure 26.3(a) repeats the flow network G and flow f of Figure 26.1(b), and Figure 26.3(b) shows the corresponding residual network G_f.

The edges in E_f are either edges in E or their reversals. If $f(u, v) < c(u, v)$ for an edge $(u, v) \in E$, then $c_f(u, v) = c(u, v) - f(u, v) > 0$ and $(u, v) \in E_f$. If $f(u, v) > 0$ for an edge $(u, v) \in E$, then $f(v, u) < 0$. In this case, $c_f(v, u) = c(v, u) - f(v, u) > 0$, and so $(v, u) \in E_f$. If neither (u, v) nor (v, u) appears in the original network, then $c(u, v) = c(v, u) = 0$, $f(u, v) = f(v, u) = 0$ (by Exercise 26.1-1), and $c_f(u, v) = c_f(v, u) = 0$. We conclude that an edge (u, v) can appear in a residual network only if at least one of (u, v) and (v, u) appears in the original network, and thus

$$|E_f| \le 2\,|E| \ .$$

Observe that the residual network G_f is itself a flow network with capacities given by c_f. The following lemma shows how a flow in a residual network relates to a flow in the original flow network.

Lemma 26.2
Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the flow sum $f + f'$ defined by equation (26.4) is a flow in G with value $|f + f'| = |f| + |f'|$.

Proof We must verify that skew symmetry, the capacity constraints, and flow conservation are obeyed. For skew symmetry, note that for all $u, v \in V$, we have

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &= -f(v, u) - f'(v, u) \\ &= -(f(v, u) + f'(v, u)) \\ &= -(f + f')(v, u) .\end{aligned}$$

For the capacity constraints, note that $f'(u, v) \le c_f(u, v)$ for all $u, v \in V$. By equation (26.5), therefore,

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &\le f(u, v) + (c(u, v) - f(u, v)) \\ &= c(u, v) .\end{aligned}$$

For flow conservation, note that for all $u \in V - \{s, t\}$, we have

$$\begin{aligned}\sum_{v \in V} (f + f')(u, v) &= \sum_{v \in V} (f(u, v) + f'(u, v)) \\ &= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) \\ &= 0 + 0 \\ &= 0 .\end{aligned}$$

Finally, we have

$$\begin{aligned}|f + f'| &= \sum_{v \in V} (f + f')(s, v) \\ &= \sum_{v \in V} (f(s, v) + f'(s, v)) \\ &= \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) \\ &= |f| + |f'| .\end{aligned}$$

■

Augmenting paths

Given a flow network $G = (V, E)$ and a flow f, an ***augmenting path*** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, each edge (u, v) on an augmenting path admits some additional positive flow from u to v without violating the capacity constraint on the edge.

The shaded path in Figure 26.3(b) is an augmenting path. Treating the residual network G_f in the figure as a flow network, we can increase the flow through each edge of this path by up to 4 units without violating a capacity constraint, since the smallest residual capacity on this path is $c_f(v_2, v_3) = 4$. We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the ***residual capacity*** of p, given by

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\} \ .$$

The following lemma, whose proof is left as Exercise 26.2-7, makes the above argument more precise.

Lemma 26.3
Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbf{R}$ by

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p \ , \\ -c_f(p) & \text{if } (v, u) \text{ is on } p \ , \\ 0 & \text{otherwise} \ . \end{cases} \qquad (26.6)$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$. ■

The following corollary shows that if we add f_p to f, we get another flow in G whose value is closer to the maximum. Figure 26.3(c) shows the result of adding f_p in Figure 26.3(b) to f from Figure 26.3(a).

Corollary 26.4
Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (26.6). Define a function $f' : V \times V \to \mathbf{R}$ by $f' = f + f_p$. Then f' is a flow in G with value $|f'| = |f| + |f_p| > |f|$.

Proof Immediate from Lemmas 26.2 and 26.3. ■

Cuts of flow networks

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until a maximum flow has been found. The max-flow min-cut theorem, which

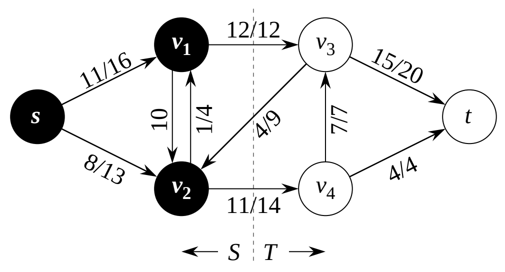


Figure 26.4 A cut (S, T) in the flow network of Figure 26.1(b), where $S = \{s, v_1, v_2\}$ and $T = \{v_3, v_4, t\}$. The vertices in S are black, and the vertices in T are white. The net flow across (S, T) is $f(S, T) = 19$, and the capacity is $c(S, T) = 26$.

we shall prove shortly, tells us that a flow is maximum if and only if its residual network contains no augmenting path. To prove this theorem, though, we must first explore the notion of a cut of a flow network.

A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. (This definition is similar to the definition of "cut" that we used for minimum spanning trees in Chapter 23, except that here we are cutting a directed graph rather than an undirected graph, and we insist that $s \in S$ and $t \in T$.) If f is a flow, then the ***net flow*** across the cut (S, T) is defined to be $f(S, T)$. The ***capacity*** of the cut (S, T) is $c(S, T)$. A ***minimum cut*** of a network is a cut whose capacity is minimum over all cuts of the network.

Figure 26.4 shows the cut $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ in the flow network of Figure 26.1(b). The net flow across this cut is

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) &= 12 + (-4) + 11 \\ &= 19 , \end{aligned}$$

and its capacity is

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26 . \end{aligned}$$

Observe that the net flow across a cut can include negative flows between vertices, but that the capacity of a cut is composed entirely of nonnegative values. In other words, the net flow across a cut (S, T) consists of positive flows in both directions; positive flow from S to T is added while positive flow from T to S is subtracted. On the other hand, the capacity of a cut (S, T) is computed only from edges going from S to T. Edges going from T to S are not included in the computation of $c(S, T)$.

The following lemma shows that the net flow across any cut is the same, and it equals the value of the flow.

Lemma 26.5
Let f be a flow in a flow network G with source s and sink t, and let (S, T) be a cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Proof Noting that $f(S - s, V) = 0$ by flow conservation, we have

$$\begin{aligned}
f(S, T) &= f(S, V) - f(S, S) && \text{(by Lemma 26.1, part (3))} \\
&= f(S, V) && \text{(by Lemma 26.1, part (1))} \\
&= f(s, V) + f(S - s, V) && \text{(by Lemma 26.1, part (3))} \\
&= f(s, V) && \text{(since } f(S - s, V) = 0\text{)} \\
&= |f| \ . && \blacksquare
\end{aligned}$$

An immediate corollary to Lemma 26.5 is the result we proved earlier—equation (26.3)—that the value of a flow is the total flow into the sink.

Another corollary to Lemma 26.5 shows how cut capacities can be used to bound the value of a flow.

Corollary 26.6
The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Proof Let (S, T) be any cut of G and let f be any flow. By Lemma 26.5 and the capacity constraints,

$$\begin{aligned}
|f| &= f(S, T) \\
&= \sum_{u \in S} \sum_{v \in T} f(u, v) \\
&\le \sum_{u \in S} \sum_{v \in T} c(u, v) \\
&= c(S, T) \ . \qquad \blacksquare
\end{aligned}$$

An immediate consequence of Corollary 26.6 is that the maximum flow in a network is bounded above by the capacity of a minimum cut of the network. The important max-flow min-cut theorem, which we now state and prove, says that the value of a maximum flow is in fact equal to the capacity of a minimum cut.

Theorem 26.7 (Max-flow min-cut theorem)
If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Proof (1) $\Rightarrow$ (2): Suppose for the sake of contradiction that f is a maximum flow in G but that G_f has an augmenting path p. Then, by Corollary 26.4, the flow sum $f + f_p$, where f_p is given by equation (26.6), is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.

(2) $\Rightarrow$ (3): Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$

and $T = V - S$. The partition (S, T) is a cut: we have $s \in S$ trivially and $t \notin S$ because there is no path from s to t in G_f. For each pair of vertices u and v such that $u \in S$ and $v \in T$, we have $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$, which would place v in set S. By Lemma 26.5, therefore, $|f| = f(S, T) = c(S, T)$.

(3) $\Rightarrow$ (1): By Corollary 26.6, $|f| \leq c(S, T)$ for all cuts (S, T). The condition $|f| = c(S, T)$ thus implies that f is a maximum flow. ■

The basic Ford-Fulkerson algorithm

In each iteration of the Ford-Fulkerson method, we find *some* augmenting path p and increase the flow f on each edge of p by the residual capacity $c_f(p)$. The following implementation of the method computes the maximum flow in a graph $G = (V, E)$ by updating the flow $f[u, v]$ between each pair u, v of vertices that are connected by an edge.[1] If u and v are not connected by an edge in either direction, we assume implicitly that $f[u, v] = 0$. The capacities $c(u, v)$ are assumed to be given along with the graph, and $c(u, v) = 0$ if $(u, v) \notin E$. The residual capacity $c_f(u, v)$ is computed in accordance with the formula (26.5). The expression $c_f(p)$ in the code is actually just a temporary variable that stores the residual capacity of the path p.

[1]We use square brackets when we treat an identifier—such as f—as a mutable field, and we use parentheses when we treat it as a function.

```
FORD-FULKERSON($G, s, t$)
1  for each edge $(u, v) \in E[G]$
2       do $f[u, v] \leftarrow 0$
3          $f[v, u] \leftarrow 0$
4  while there exists a path $p$ from $s$ to $t$ in the residual network $G_f$
5       do $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v)$ is in $p\}$
6          for each edge $(u, v)$ in $p$
7               do $f[u, v] \leftarrow f[u, v] + c_f(p)$
8                  $f[v, u] \leftarrow -f[u, v]$
```

The FORD-FULKERSON algorithm simply expands on the FORD-FULKERSON-METHOD pseudocode given earlier. Figure 26.5 shows the result of each iteration in a sample run. Lines 1–3 initialize the flow f to 0. The **while** loop of lines 4–8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$. When no augmenting paths exist, the flow f is a maximum flow.

Analysis of Ford-Fulkerson

The running time of FORD-FULKERSON depends on how the augmenting path p in line 4 is determined. If it is chosen poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value.[2] If the augmenting path is chosen by using a breadth-first search (which we saw in Section 22.2), however, the algorithm runs in polynomial time. Before proving this result, however, we obtain a simple bound for the case in which the augmenting path is chosen arbitrarily and all capacities are integers.

Most often in practice, the maximum-flow problem arises with integral capacities. If the capacities are rational numbers, an appropriate scaling transformation can be used to make them all integral. Under this assumption, a straightforward implementation of FORD-FULKERSON runs in time $O(E\ |f^*|)$, where f^* is the maximum flow found by the algorithm. The analysis is as follows. Lines 1–3 take time $\Theta(E)$. The **while** loop of lines 4–8 is executed at most $|f^*|$ times, since the flow value increases by at least one unit in each iteration.

The work done within the **while** loop can be made efficient if we efficiently manage the data structure used to implement the network $G = (V, E)$. Let us assume that we keep a data structure corresponding to a directed graph $G' = (V, E')$, where $E' = \{(u, v) : (u, v) \in E \text{ or } (v, u) \in E\}$. Edges in the network G are also

[2]The Ford-Fulkerson method might fail to terminate only if edge capacities are irrational numbers.

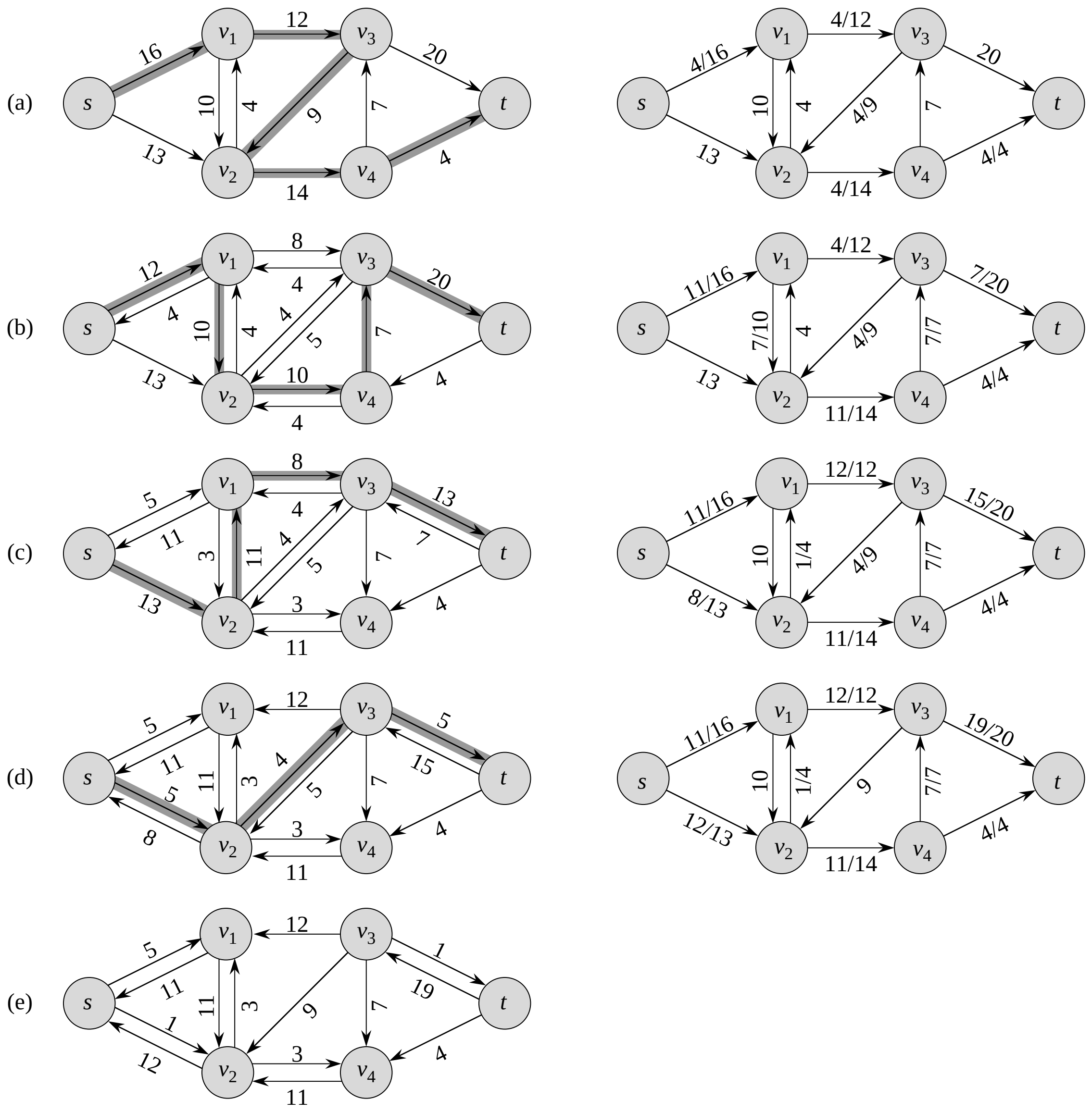


Figure 26.5 The execution of the basic Ford-Fulkerson algorithm. **(a)–(d)** Successive iterations of the **while** loop. The left side of each part shows the residual network G_f from line 4 with a shaded augmenting path p. The right side of each part shows the new flow f that results from adding f_p to f. The residual network in (a) is the input network G. **(e)** The residual network at the last **while** loop test. It has no augmenting paths, and the flow f shown in (d) is therefore a maximum flow.

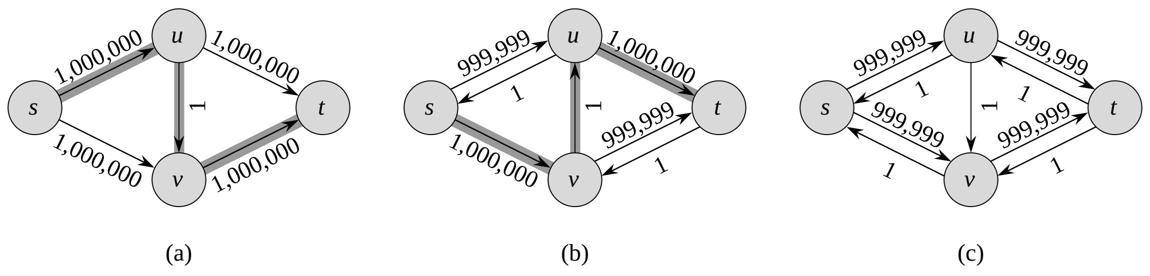


Figure 26.6 **(a)** A flow network for which FORD-FULKERSON can take $\Theta(E\,|f^*|)$ time, where f^* is a maximum flow, shown here with $|f^*| = 2{,}000{,}000$. An augmenting path with residual capacity 1 is shown. **(b)** The resulting residual network. Another augmenting path with residual capacity 1 is shown. **(c)** The resulting residual network.

edges in G', and it is therefore a simple matter to maintain capacities and flows in this data structure. Given a flow f on G, the edges in the residual network G_f consist of all edges (u, v) of G' such that $c(u, v) - f[u, v] \neq 0$. The time to find a path in a residual network is therefore $O(V + E') = O(E)$ if we use either depth-first search or breadth-first search. Each iteration of the **while** loop thus takes $O(E)$ time, making the total running time of FORD-FULKERSON $O(E\,|f^*|)$.

When the capacities are integral and the optimal flow value $|f^*|$ is small, the running time of the Ford-Fulkerson algorithm is good. Figure 26.6(a) shows an example of what can happen on a simple flow network for which $|f^*|$ is large. A maximum flow in this network has value 2,000,000: 1,000,000 units of flow traverse the path $s \to u \to t$, and another 1,000,000 units traverse the path $s \to v \to t$. If the first augmenting path found by FORD-FULKERSON is $s \to u \to v \to t$, shown in Figure 26.6(a), the flow has value 1 after the first iteration. The resulting residual network is shown in Figure 26.6(b). If the second iteration finds the augmenting path $s \to v \to u \to t$, as shown in Figure 26.6(b), the flow then has value 2. Figure 26.6(c) shows the resulting residual network. We can continue, choosing the augmenting path $s \to u \to v \to t$ in the odd-numbered iterations and the augmenting path $s \to v \to u \to t$ in the even-numbered iterations. We would perform a total of 2,000,000 augmentations, increasing the flow value by only 1 unit in each.

The Edmonds-Karp algorithm

The bound on FORD-FULKERSON can be improved if we implement the computation of the augmenting path p in line 4 with a breadth-first search, that is, if the augmenting path is a *shortest* path from s to t in the residual network, where each edge has unit distance (weight). We call the Ford-Fulkerson method so implemented the ***Edmonds-Karp algorithm***. We now prove that the Edmonds-Karp algorithm runs in $O(VE^2)$ time.

The analysis depends on the distances to vertices in the residual network G_f. The following lemma uses the notation $\delta_f(u, v)$ for the shortest-path distance from u to v in G_f, where each edge has unit distance.

Lemma 26.8
If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

Proof We will suppose that for some vertex $v \in V - \{s, t\}$, there is a flow augmentation that causes the shortest-path distance from s to v to decrease, and then we will derive a contradiction. Let f be the flow just before the first augmentation that decreases some shortest-path distance, and let f' be the flow just afterward. Let v be the vertex with the minimum $\delta_{f'}(s, v)$ whose distance was decreased by the augmentation, so that $\delta_{f'}(s, v) < \delta_f(s, v)$. Let $p = s \leadsto u \to v$ be a shortest path from s to v in $G_{f'}$, so that $(u, v) \in E_{f'}$ and

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1 . \tag{26.7}$$

Because of how we chose v, we know that the distance label of vertex u did not decrease, i.e.,

$$\delta_{f'}(s, u) \geq \delta_f(s, u) . \tag{26.8}$$

We claim that $(u, v) \notin E_f$. Why? If we had $(u, v) \in E_f$, then we would also have

$$\begin{aligned}
\delta_f(s, v) &\leq \delta_f(s, u) + 1 && \text{(by Lemma 24.10, the triangle inequality)} \\
&\leq \delta_{f'}(s, u) + 1 && \text{(by inequality (26.8))} \\
&= \delta_{f'}(s, v) && \text{(by equation (26.7))} ,
\end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$.

How can we have $(u, v) \notin E_f$ and $(u, v) \in E_{f'}$? The augmentation must have increased the flow from v to u. The Edmonds-Karp algorithm always augments flow along shortest paths, and therefore the shortest path from s to u in G_f has (v, u) as its last edge. Therefore,

$$\begin{aligned}
\delta_f(s, v) &= \delta_f(s, u) - 1 \\
&\leq \delta_{f'}(s, u) - 1 && \text{(by inequality (26.8))} \\
&= \delta_{f'}(s, v) - 2 && \text{(by equation (26.7))} ,
\end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$. We conclude that our assumption that such a vertex v exists is incorrect. ■

The next theorem bounds the number of iterations of the Edmonds-Karp algorithm.

Theorem 26.9
If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

Proof We say that an edge (u, v) in a residual network G_f is ***critical*** on an augmenting path p if the residual capacity of p is the residual capacity of (u, v), that is, if $c_f(p) = c_f(u, v)$. After we have augmented flow along an augmenting path, any critical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical. We will show that each of the $|E|$ edges can become critical at most $|V|/2$ times.

Let u and v be vertices in V that are connected by an edge in E. Since augmenting paths are shortest paths, when (u, v) is critical for the first time, we have

$$\delta_f(s, v) = \delta_f(s, u) + 1 .$$

Once the flow is augmented, the edge (u, v) disappears from the residual network. It cannot reappear later on another augmenting path until after the flow from u to v is decreased, which occurs only if (v, u) appears on an augmenting path. If f' is the flow in G when this event occurs, then we have

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 .$$

Since $\delta_f(s, v) \leq \delta_{f'}(s, v)$ by Lemma 26.8, we have

$$\begin{aligned} \delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\geq \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2 . \end{aligned}$$

Consequently, from the time (u, v) becomes critical to the time when it next becomes critical, the distance of u from the source increases by at least 2. The distance of u from the source is initially at least 0. The intermediate vertices on a shortest path from s to u cannot contain s, u, or t (since (u, v) on the critical path implies that $u \neq t$). Therefore, until u becomes unreachable from the source, if ever, its distance is at most $|V| - 2$. Thus, after the first time that (u, v) becomes critical, it can become critical at most $(|V| - 2)/2 = |V|/2 - 1$ times more, for a total of at most $|V|/2$ times. Since there are $O(E)$ pairs of vertices that can have an edge between them in a residual graph, the total number of critical edges during the entire execution of the Edmonds-Karp algorithm is $O(VE)$. Each augmenting path has at least one critical edge, and hence the theorem follows. ■

Since each iteration of FORD-FULKERSON can be implemented in $O(E)$ time when the augmenting path is found by breadth-first search, the total running time of the Edmonds-Karp algorithm is $O(VE^2)$. We shall see that push-relabel algorithms can yield even better bounds. The algorithm of Section 26.4 gives a method for achieving an $O(V^2E)$ running time, which forms the basis for the $O(V^3)$-time algorithm of Section 26.5.

Exercises

26.2-1
In Figure 26.1(b), what is the flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

26.2-2
Show the execution of the Edmonds-Karp algorithm on the flow network of Figure 26.1(a).

26.2-3
In the example of Figure 26.5, what is the minimum cut corresponding to the maximum flow shown? Of the augmenting paths appearing in the example, which two cancel flow?

26.2-4
Prove that for any pair of vertices u and v and any capacity and flow functions c and f, we have $c_f(u,v) + c_f(v,u) = c(u,v) + c(v,u)$.

26.2-5
Recall that the construction in Section 26.1 that converts a multisource, multisink flow network into a single-source, single-sink network adds edges with infinite capacity. Prove that any flow in the resulting network has a finite value if the edges of the original multisource, multisink network have finite capacity.

26.2-6
Suppose that each source s_i in a multisource, multisink problem produces exactly p_i units of flow, so that $f(s_i, V) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $f(V, t_j) = q_j$, where $\sum_i p_i = \sum_j q_j$. Show how to convert the problem of finding a flow f that obeys these additional constraints into the problem of finding a maximum flow in a single-source, single-sink flow network.

26.2-7
Prove Lemma 26.3.

26.2-8

Show that a maximum flow in a network $G = (V, E)$ can always be found by a sequence of at most $|E|$ augmenting paths. (*Hint:* Determine the paths *after* finding the maximum flow.)

26.2-9

The ***edge connectivity*** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how the edge connectivity of an undirected graph $G = (V, E)$ can be determined by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

26.2-10

Suppose that a flow network $G = (V, E)$ has symmetric edges, that is, $(u, v) \in E$ if and only if $(v, u) \in E$. Show that the Edmonds-Karp algorithm terminates after at most $|V| \, |E| / 4$ iterations. (*Hint:* For any edge (u, v), consider how both $\delta(s, u)$ and $\delta(v, t)$ change between times at which (u, v) is critical.)

26.3 Maximum bipartite matching

Some combinatorial problems can easily be cast as maximum-flow problems. The multiple-source, multiple-sink maximum-flow problem from Section 26.1 gave us one example. There are other combinatorial problems that seem on the surface to have little to do with flow networks, but can in fact be reduced to maximum-flow problems. This section presents one such problem: finding a maximum matching in a bipartite graph (see Section B.4). In order to solve this problem, we shall take advantage of an integrality property provided by the Ford-Fulkerson method. We shall also see that the Ford-Fulkerson method can be made to solve the maximum-bipartite-matching problem on a graph $G = (V, E)$ in $O(VE)$ time.

The maximum-bipartite-matching problem

Given an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is ***matched*** by matching M if some edge in M is incident on v; otherwise, v is ***unmatched***. A ***maximum matching*** is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M'|$. In this section, we shall restrict our attention to finding maximum matchings in bipartite graphs. We assume that the vertex set can be partitioned into $V = L \cup R$,

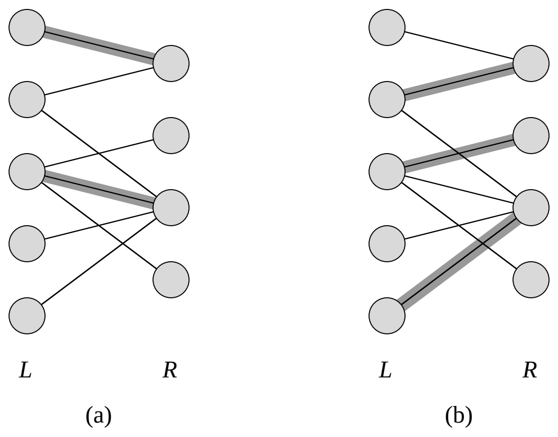


Figure 26.7 A bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$. **(a)** A matching with cardinality 2. **(b)** A maximum matching with cardinality 3.

where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge. Figure 26.7 illustrates the notion of a matching.

The problem of finding a maximum matching in a bipartite graph has many practical applications. As an example, we might consider matching a set L of machines with a set R of tasks to be performed simultaneously. We take the presence of edge (u, v) in E to mean that a particular machine $u \in L$ is capable of performing a particular task $v \in R$. A maximum matching provides work for as many machines as possible.

Finding a maximum bipartite matching

We can use the Ford-Fulkerson method to find a maximum matching in an undirected bipartite graph $G = (V, E)$ in time polynomial in $|V|$ and $|E|$. The trick is to construct a flow network in which flows correspond to matchings, as shown in Figure 26.8. We define the ***corresponding flow network*** $G' = (V', E')$ for the bipartite graph G as follows. We let the source s and sink t be new vertices not in V, and we let $V' = V \cup \{s, t\}$. If the vertex partition of G is $V = L \cup R$, the directed edges of G' are the edges of E, directed from L to R, along with $|V|$ new edges:

$$\begin{aligned} E' &= \{(s, u) : u \in L\} \\ &\quad \cup \{(u, v) : u \in L,\ v \in R,\ \text{and } (u, v) \in E\} \\ &\quad \cup \{(v, t) : v \in R\}\ . \end{aligned}$$

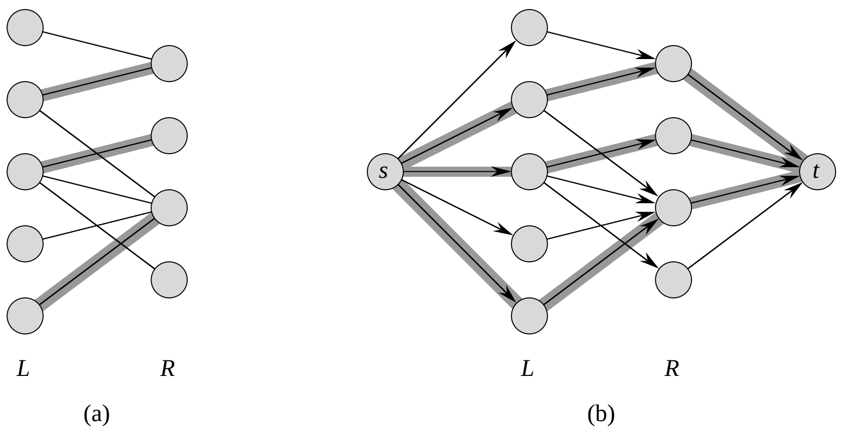


Figure 26.8 The flow network corresponding to a bipartite graph. **(a)** The bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$ from Figure 26.7. A maximum matching is shown by shaded edges. **(b)** The corresponding flow network G' with a maximum flow shown. Each edge has unit capacity. Shaded edges have a flow of 1, and all other edges carry no flow. The shaded edges from L to R correspond to those in a maximum matching of the bipartite graph.

To complete the construction, we assign unit capacity to each edge in E'. Since each vertex in V has at least one incident edge, $|E| \geq |V|/2$. Thus, $|E| \leq |E'| = |E| + |V| \leq 3|E|$, and so $|E'| = \Theta(E)$.

The following lemma shows that a matching in G corresponds directly to a flow in G's corresponding flow network G'. We say that a flow f on a flow network $G = (V, E)$ is ***integer-valued*** if $f(u, v)$ is an integer for all $(u, v) \in V \times V$.

Lemma 26.10

Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let $G' = (V', E')$ be its corresponding flow network. If M is a matching in G, then there is an integer-valued flow f in G' with value $|f| = |M|$. Conversely, if f is an integer-valued flow in G', then there is a matching M in G with cardinality $|M| = |f|$.

Proof We first show that a matching M in G corresponds to an integer-valued flow f in G'. Define f as follows. If $(u, v) \in M$, then $f(s, u) = f(u, v) = f(v, t) = 1$ and $f(u, s) = f(v, u) = f(t, v) = -1$. For all other edges $(u, v) \in E'$, we define $f(u, v) = 0$. It is simple to verify that f satisfies skew symmetry, the capacity constraints, and flow conservation.

Intuitively, each edge $(u, v) \in M$ corresponds to 1 unit of flow in G' that traverses the path $s \to u \to v \to t$. Moreover, the paths induced by edges in M are vertex-disjoint, except for s and t. The net flow across cut $(L \cup \{s\}, R \cup \{t\})$ is equal to $|M|$; thus, by Lemma 26.5, the value of the flow is $|f| = |M|$.

To prove the converse, let f be an integer-valued flow in G', and let

$$M = \{(u, v) : u \in L,\ v \in R,\ \text{and}\ f(u, v) > 0\} .$$

Each vertex $u \in L$ has only one entering edge, namely (s, u), and its capacity is 1. Thus, each $u \in L$ has at most one unit of positive flow entering it, and if one unit of positive flow does enter, by flow conservation, one unit of positive flow must leave. Furthermore, since f is integer-valued, for each $u \in L$, the one unit of flow can enter on at most one edge and can leave on at most one edge. Thus, one unit of positive flow enters u if and only if there is exactly one vertex $v \in R$ such that $f(u, v) = 1$, and at most one edge leaving each $u \in L$ carries positive flow. A symmetric argument can be made for each $v \in R$. The set M is therefore a matching.

To see that $|M| = |f|$, observe that for every matched vertex $u \in L$, we have $f(s, u) = 1$, and for every edge $(u, v) \in E - M$, we have $f(u, v) = 0$. Consequently,

$$\begin{aligned} |M| &= f(L, R) \\ &= f(L, V') - f(L, L) - f(L, s) - f(L, t) \quad \text{(by Lemma 26.1)} . \end{aligned}$$

We can simplify the above expression considerably. Flow conservation implies that $f(L, V') = 0$; Lemma 26.1 implies that $f(L, L) = 0$; skew symmetry implies that $-f(L, s) = f(s, L)$; and because there are no edges from L to t, we have $f(L, t) = 0$. Thus,

$$\begin{aligned} |M| &= f(s, L) \\ &= f(s, V') \quad \text{(since all edges out of } s \text{ go to } L) \\ &= |f| \quad \text{(by the definition of } |f|) . \end{aligned}$$

■

Based on Lemma 26.10, we would like to conclude that a maximum matching in a bipartite graph G corresponds to a maximum flow in its corresponding flow network G', and we can therefore compute a maximum matching in G by running a maximum-flow algorithm on G'. The only hitch in this reasoning is that the maximum-flow algorithm might return a flow in G' for which some $f(u, v)$ is not an integer, even though the flow value $|f|$ must be an integer. The following theorem shows that if we use the Ford-Fulkerson method, this difficulty cannot arise.

Theorem 26.11 (Integrality theorem)

If the capacity function c takes on only integral values, then the maximum flow f produced by the Ford-Fulkerson method has the property that $|f|$ is integer-valued. Moreover, for all vertices u and v, the value of $f(u, v)$ is an integer.

Proof The proof is by induction on the number of iterations. We leave it as Exercise 26.3-2. ■

We can now prove the following corollary to Lemma 26.10.

Corollary 26.12
The cardinality of a maximum matching M in a bipartite graph G equals the value of a maximum flow f in its corresponding flow network G'.

Proof We use the nomenclature from Lemma 26.10. Suppose that M is a maximum matching in G and that the corresponding flow f in G' is not maximum. Then there is a maximum flow f' in G' such that $|f'| > |f|$. Since the capacities in G' are integer-valued, by Theorem 26.11, we can assume that f' is integer-valued. Thus, f' corresponds to a matching M' in G with cardinality $|M'| = |f'| > |f| = |M|$, contradicting our assumption that M is a maximum matching. In a similar manner, we can show that if f is a maximum flow in G', its corresponding matching is a maximum matching on G. ■

Thus, given a bipartite undirected graph G, we can find a maximum matching by creating the flow network G', running the Ford-Fulkerson method, and directly obtaining a maximum matching M from the integer-valued maximum flow f found. Since any matching in a bipartite graph has cardinality at most $\min(L, R) = O(V)$, the value of the maximum flow in G' is $O(V)$. We can therefore find a maximum matching in a bipartite graph in time $O(VE') = O(VE)$, since $|E'| = \Theta(E)$.

Exercises

26.3-1
Run the Ford-Fulkerson algorithm on the flow network in Figure 26.8(b) and show the residual network after each flow augmentation. Number the vertices in L top to bottom from 1 to 5 and in R top to bottom from 6 to 9. For each iteration, pick the augmenting path that is lexicographically smallest.

26.3-2
Prove Theorem 26.11.

26.3-3
Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.

26.3-4 $\star$
A ***perfect matching*** is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the ***neighborhood*** of X as

$$N(X) = \{y \in V : (x, y) \in E \text{ for some } x \in X\} ,$$

that is, the set of vertices adjacent to some member of X. Prove ***Hall's theorem***: there exists a perfect matching in G if and only if $|A| \leq |N(A)|$ for every subset $A \subseteq L$.

26.3-5 $\star$
We say that a bipartite graph $G = (V, E)$, where $V = L \cup R$, is ***d-regular*** if every vertex $v \in V$ has degree exactly d. Every d-regular bipartite graph has $|L| = |R|$. Prove that every d-regular bipartite graph has a matching of cardinality $|L|$ by arguing that a minimum cut of the corresponding flow network has capacity $|L|$.

$\star$ 26.4 Push-relabel algorithms

In this section, we present the "push-relabel" approach to computing maximum flows. To date, many of the asymptotically fastest maximum-flow algorithms are push-relabel algorithms, and the fastest actual implementations of maximum-flow algorithms are based on the push-relabel method. Other flow problems, such as the minimum-cost flow problem, can be solved efficiently by push-relabel methods. This section introduces Goldberg's "generic" maximum-flow algorithm, which has a simple implementation that runs in $O(V^2E)$ time, thereby improving upon the $O(VE^2)$ bound of the Edmonds-Karp algorithm. Section 26.5 refines the generic algorithm to obtain another push-relabel algorithm that runs in $O(V^3)$ time.

Push-relabel algorithms work in a more localized manner than the Ford-Fulkerson method. Rather than examine the entire residual network to find an augmenting path, push-relabel algorithms work on one vertex at a time, looking only at the vertex's neighbors in the residual network. Furthermore, unlike the Ford-Fulkerson method, push-relabel algorithms do not maintain the flow-conservation property throughout their execution. They do, however, maintain a ***preflow***, which is a function $f : V \times V \to \mathbf{R}$ that satisfies skew symmetry, capacity constraints, and the following relaxation of flow conservation: $f(V, u) \geq 0$ for all vertices $u \in V - \{s\}$. We call this quantity the ***excess flow*** into vertex u, given by

$$e(u) = f(V, u) . \tag{26.9}$$

We say that a vertex $u \in V - \{s, t\}$ is ***overflowing*** if $e(u) > 0$.

We shall start this section by describing the intuition behind the push-relabel method. We shall then investigate the two operations employed by the method: "pushing" preflow and "relabeling" a vertex. Finally, we shall present a generic push-relabel algorithm and analyze its correctness and running time.

Intuition

The intuition behind the push-relabel method is probably best understood in terms of fluid flows: we consider a flow network $G = (V, E)$ to be a system of interconnected pipes of given capacities. Applying this analogy to the Ford-Fulkerson method, we might say that each augmenting path in the network gives rise to an additional stream of fluid, with no branch points, flowing from the source to the sink. The Ford-Fulkerson method iteratively adds more streams of flow until no more can be added.

The generic push-relabel algorithm has a rather different intuition. As before, directed edges correspond to pipes. Vertices, which are pipe junctions, have two interesting properties. First, to accommodate excess flow, each vertex has an outflow pipe leading to an arbitrarily large reservoir that can accumulate fluid. Second, each vertex, its reservoir, and all its pipe connections are on a platform whose height increases as the algorithm progresses.

Vertex heights determine how flow is pushed: we only push flow downhill, that is, from a higher vertex to a lower vertex. The flow from a lower vertex to a higher vertex may be positive, but operations that push flow only push it downhill. The height of the source is fixed at $|V|$, and the height of the sink is fixed at 0. All other vertex heights start at 0 and increase with time. The algorithm first sends as much flow as possible downhill from the source toward the sink. The amount it sends is exactly enough to fill each outgoing pipe from the source to capacity; that is, it sends the capacity of the cut $(s, V - s)$. When flow first enters an intermediate vertex, it collects in the vertex's reservoir. From there, it is eventually pushed downhill.

It may eventually happen that the only pipes that leave a vertex u and are not already saturated with flow connect to vertices that are on the same level as u or are uphill from u. In this case, to rid an overflowing vertex u of its excess flow, we must increase its height—an operation called "relabeling" vertex u. Its height is increased to one unit more than the height of the lowest of its neighbors to which it has an unsaturated pipe. After a vertex is relabeled, therefore, there is at least one outgoing pipe through which more flow can be pushed.

Eventually, all the flow that can possibly get through to the sink has arrived there. No more can arrive, because the pipes obey the capacity constraints; the amount of flow across any cut is still limited by the capacity of the cut. To make the preflow a "legal" flow, the algorithm then sends the excess collected in the reservoirs of overflowing vertices back to the source by continuing to relabel vertices to above the fixed height $|V|$ of the source. As we shall see, once all the reservoirs have been emptied, the preflow is not only a “legal” flow, it is also a maximum flow.

The basic operations

From the preceding discussion, we see that there are two basic operations performed by a push-relabel algorithm: pushing flow excess from a vertex to one of its neighbors and relabeling a vertex. The applicability of these operations depends on the heights of vertices, which we now define precisely.

Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a preflow in G. A function $h : V \to \mathbf{N}$ is a ***height function***[3] if $h(s) = |V|$, $h(t) = 0$, and

$$h(u) \le h(v) + 1$$

for every residual edge $(u, v) \in E_f$. We immediately obtain the following lemma.

Lemma 26.13
Let $G = (V, E)$ be a flow network, let f be a preflow in G, and let h be a height function on V. For any two vertices $u, v \in V$, if $h(u) > h(v) + 1$, then (u, v) is not an edge in the residual graph. ■

The push operation
The basic operation PUSH(u, v) can be applied if u is an overflowing vertex, $c_f(u, v) > 0$, and $h(u) = h(v) + 1$. The pseudocode below updates the preflow f in an implied network $G = (V, E)$. It assumes that residual capacities can also be computed in constant time given c and f. The excess flow stored at a vertex u is maintained as the attribute $e[u]$, and the height of u is maintained as the attribute $h[u]$. The expression $d_f(u, v)$ is a temporary variable that stores the amount of flow that can be pushed from u to v.

[3]In the literature, a height function is typically called a “distance function,” and the height of a vertex is called a “distance label.” We use the term “height” because it is more suggestive of the intuition behind the algorithm. We retain the use of the term “relabel” to refer to the operation that increases the height of a vertex. The height of a vertex is related to its distance from the sink t, as would be found in a breadth-first search of the transpose G^{T}.

```
PUSH$(u, v)$
1  ▷ Applies when: $u$ is overflowing, $c_f(u, v) > 0$, and $h[u] = h[v] + 1$.
2  ▷ Action: Push $d_f(u, v) = \min(e[u], c_f(u, v))$ units of flow from $u$ to $v$.
3  $d_f(u, v) \leftarrow \min(e[u], c_f(u, v))$
4  $f[u, v] \leftarrow f[u, v] + d_f(u, v)$
5  $f[v, u] \leftarrow -f[u, v]$
6  $e[u] \leftarrow e[u] - d_f(u, v)$
7  $e[v] \leftarrow e[v] + d_f(u, v)$
```

The code for PUSH operates as follows. Vertex u is assumed to have a positive excess $e[u]$, and the residual capacity of (u, v) is positive. Thus, we can increase the flow from u to v by $d_f(u, v) = \min(e[u], c_f(u, v))$ without causing $e[u]$ to become negative or the capacity $c(u, v)$ to be exceeded. Line 3 computes the value $d_f(u, v)$, and we update f in lines 4–5 and e in lines 6–7. Thus, if f is a preflow before PUSH is called, it remains a preflow afterward.

Observe that nothing in the code for PUSH depends on the heights of u and v, yet we prohibit it from being invoked unless $h[u] = h[v] + 1$. Thus, excess flow is pushed downhill only by a height differential of 1. By Lemma 26.13, no residual edges exist between two vertices whose heights differ by more than 1, and thus, as long as the attribute h is indeed a height function, there is nothing to be gained by allowing flow to be pushed downhill by a height differential of more than 1.

We call the operation PUSH(u, v) a ***push*** from u to v. If a push operation applies to some edge (u, v) leaving a vertex u, we also say that the push operation applies to u. It is a ***saturating push*** if edge (u, v) becomes ***saturated*** ($c_f(u, v) = 0$ afterward); otherwise, it is a ***nonsaturating push***. If an edge is saturated, it does not appear in the residual network. A simple lemma characterizes one result of a nonsaturating push.

Lemma 26.14
After a nonsaturating push from u to v, the vertex u is no longer overflowing.

Proof Since the push was nonsaturating, the amount of flow $d_f(u, v)$ actually pushed must equal $e[u]$ prior to the push. Since $e[u]$ is reduced by this amount, it becomes 0 after the push. ■

The relabel operation

The basic operation RELABEL(u) applies if u is overflowing and if $h[u] \le h[v]$ for all edges $(u, v) \in E_f$. In other words, we can relabel an overflowing vertex u if for every vertex v for which there is residual capacity from u to v, flow cannot be pushed from u to v because v is not downhill from u. (Recall that by definition, neither the source s nor the sink t can be overflowing, so neither s nor t can be relabeled.)

```
RELABEL($u$)
1  ▷ Applies when: $u$ is overflowing and for all $v \in V$ such that $(u, v) \in E_f$,
                   we have $h[u] \le h[v]$.
2  ▷ Action: Increase the height of $u$.
3  $h[u] \leftarrow 1 + \min\{h[v] : (u, v) \in E_f\}$
```

When we call the operation RELABEL(u), we say that vertex u is ***relabeled***. Note that when u is relabeled, E_f must contain at least one edge that leaves u, so that the minimization in the code is over a nonempty set. This property follows from the assumption that u is overflowing. Since $e[u] > 0$, we have $e[u] = f(V, u) > 0$, and hence there must be at least one vertex v such that $f[v, u] > 0$. But then,

$$\begin{aligned} c_f(u, v) &= c(u, v) - f[u, v] \\ &= c(u, v) + f[v, u] \\ &> 0 , \end{aligned}$$

which implies that $(u, v) \in E_f$. The operation RELABEL(u) thus gives u the greatest height allowed by the constraints on height functions.

The generic algorithm

The generic push-relabel algorithm uses the following subroutine to create an initial preflow in the flow network.

```
INITIALIZE-PREFLOW($G, s$)
 1  for each vertex $u \in V[G]$
 2       do $h[u] \leftarrow 0$
 3          $e[u] \leftarrow 0$
 4  for each edge $(u, v) \in E[G]$
 5       do $f[u, v] \leftarrow 0$
 6          $f[v, u] \leftarrow 0$
 7  $h[s] \leftarrow |V[G]|$
 8  for each vertex $u \in Adj[s]$
 9       do $f[s, u] \leftarrow c(s, u)$
10          $f[u, s] \leftarrow -c(s, u)$
11          $e[u] \leftarrow c(s, u)$
12          $e[s] \leftarrow e[s] - c(s, u)$
```

INITIALIZE-PREFLOW creates an initial preflow f defined by

$$f[u,v] = \begin{cases} c(u,v) & \text{if } u = s\ , \\ -c(v,u) & \text{if } v = s\ , \\ 0 & \text{otherwise}\ . \end{cases} \tag{26.10}$$

That is, each edge leaving the source s is filled to capacity, and all other edges carry no flow. For each vertex v adjacent to the source, we initially have $e[v] = c(s,v)$, and $e[s]$ is initialized to the negative of the sum of these capacities. The generic algorithm also begins with an initial height function h, given by

$$h[u] = \begin{cases} |V| & \text{if } u = s\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

This is a height function because the only edges (u,v) for which $h[u] > h[v] + 1$ are those for which $u = s$, and those edges are saturated, which means that they are not in the residual network.

Initialization, followed by a sequence of push and relabel operations, executed in no particular order, yields the GENERIC-PUSH-RELABEL algorithm:

```
GENERIC-PUSH-RELABEL(G)
1  INITIALIZE-PREFLOW(G, s)
2  while there exists an applicable push or relabel operation
3       do select an applicable push or relabel operation and perform it
```

The following lemma tells us that as long as an overflowing vertex exists, at least one of the two basic operations applies.

Lemma 26.15 (An overflowing vertex can be either pushed or relabeled)
Let $G = (V, E)$ be a flow network with source s and sink t, let f be a preflow, and let h be any height function for f. If u is any overflowing vertex, then either a push or relabel operation applies to it.

Proof For any residual edge (u,v), we have $h(u) \le h(v)+1$ because h is a height function. If a push operation does not apply to u, then for all residual edges (u,v), we must have $h(u) < h(v) + 1$, which implies $h(u) \le h(v)$. Thus, a relabel operation can be applied to u. ■

Correctness of the push-relabel method

To show that the generic push-relabel algorithm solves the maximum-flow problem, we shall first prove that if it terminates, the preflow f is a maximum flow. We shall later prove that it terminates. We start with some observations about the height function h.

Lemma 26.16 (Vertex heights never decrease)
During the execution of GENERIC-PUSH-RELABEL on a flow network $G = (V, E)$, for each vertex $u \in V$, the height $h[u]$ never decreases. Moreover, whenever a relabel operation is applied to a vertex u, its height $h[u]$ increases by at least 1.

Proof Because vertex heights change only during relabel operations, it suffices to prove the second statement of the lemma. If vertex u is about to be relabeled, then for all vertices v such that $(u, v) \in E_f$, we have $h[u] \leq h[v]$. Thus, $h[u] < 1 + \min\{h[v] : (u, v) \in E_f\}$, and so the operation must increase $h[u]$. ■

Lemma 26.17
Let $G = (V, E)$ be a flow network with source s and sink t. During the execution of GENERIC-PUSH-RELABEL on G, the attribute h is maintained as a height function.

Proof The proof is by induction on the number of basic operations performed. Initially, h is a height function, as we have already observed.

We claim that if h is a height function, then an operation RELABEL(u) leaves h a height function. If we look at a residual edge $(u, v) \in E_f$ that leaves u, then the operation RELABEL(u) ensures that $h[u] \leq h[v] + 1$ afterward. Now consider a residual edge (w, u) that enters u. By Lemma 26.16, $h[w] \leq h[u] + 1$ before the operation RELABEL(u) implies $h[w] < h[u] + 1$ afterward. Thus, the operation RELABEL(u) leaves h a height function.

Now, consider an operation PUSH(u, v). This operation may add the edge (v, u) to E_f, and it may remove (u, v) from E_f. In the former case, we have $h[v] = h[u] - 1 < h[u] + 1$, and so h remains a height function. In the latter case, the removal of (u, v) from the residual network removes the corresponding constraint, and h again remains a height function. ■

The following lemma gives an important property of height functions.

Lemma 26.18
Let $G = (V, E)$ be a flow network with source s and sink t, let f be a preflow in G, and let h be a height function on V. Then there is no path from the source s to the sink t in the residual network G_f.

Proof Assume for the sake of contradiction that there is a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ from s to t in G_f, where $v_0 = s$ and $v_k = t$. Without loss of generality, p is a simple path, and so $k < |V|$. For $i = 0, 1, \ldots, k - 1$, edge $(v_i, v_{i+1}) \in E_f$. Because h is a height function, $h(v_i) \leq h(v_{i+1}) + 1$ for $i = 0, 1, \ldots, k - 1$. Combining these inequalities over path p yields $h(s) \leq h(t) + k$. But because $h(t) = 0$, we have $h(s) \le k < |V|$, which contradicts the requirement that $h(s) = |V|$ in a height function. ■

We are now ready to show that if the generic push-relabel algorithm terminates, the preflow it computes is a maximum flow.

Theorem 26.19 (Correctness of the generic push-relabel algorithm)
If the algorithm GENERIC-PUSH-RELABEL terminates when run on a flow network $G = (V, E)$ with source s and sink t, then the preflow f it computes is a maximum flow for G.

Proof We use the following loop invariant:

> Each time the **while** loop test in line 2 in GENERIC-PUSH-RELABEL is executed, f is a preflow.

Initialization: INITIALIZE-PREFLOW makes f a preflow.

Maintenance: The only operations within the **while** loop of lines 2–3 are push and relabel. Relabel operations affect only height attributes and not the flow values; hence they do not affect whether f is a preflow. As argued on page 672, if f is a preflow prior to a push operation, it remains a preflow afterward.

Termination: At termination, each vertex in $V - \{s, t\}$ must have an excess of 0, because by Lemmas 26.15 and 26.17 and the invariant that f is always a preflow, there are no overflowing vertices. Therefore, f is a flow. Because h is a height function, Lemma 26.18 tells us that there is no path from s to t in the residual network G_f. By the max-flow min-cut theorem (Theorem 26.7), therefore, f is a maximum flow. ■

Analysis of the push-relabel method

To show that the generic push-relabel algorithm indeed terminates, we shall bound the number of operations it performs. Each of the three types of operations—relabels, saturating pushes, and nonsaturating pushes—is bounded separately. With knowledge of these bounds, it is a straightforward problem to construct an algorithm that runs in $O(V^2E)$ time. Before beginning the analysis, however, we prove an important lemma.

Lemma 26.20
Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a preflow in G. Then, for any overflowing vertex u, there is a simple path from u to s in the residual network G_f.

Proof For an overflowing vertex u, let $U = \{v :$ there exists a simple path from u to v in $G_f\}$, and suppose for the sake of contradiction that $s \notin U$. Let $\overline{U} = V - U$.

We claim for each pair of vertices $w \in \overline{U}$ and $v \in U$ that $f(w, v) \le 0$. Why? If $f(w, v) > 0$, then $f(v, w) < 0$, which in turn implies that $c_f(v, w) = c(v, w) - f(v, w) > 0$. Hence, there exists an edge $(v, w) \in E_f$, and therefore a simple path of the form $u \rightsquigarrow v \to w$ in G_f, contradicting our choice of w.

Thus, we must have $f(\overline{U}, U) \le 0$, since every term in this implicit summation is nonpositive, and hence

$$\begin{aligned} e(U) &= f(V, U) && \text{(by equation (26.9))} \\ &= f(\overline{U}, U) + f(U, U) && \text{(by Lemma 26.1, part (3))} \\ &= f(\overline{U}, U) && \text{(by Lemma 26.1, part (1))} \\ &\le 0 . \end{aligned}$$

(Although Lemma 26.1 applies to flows, Exercise 26.1-4 demonstrates that it does not rely on flow conservation. Hence, Lemma 26.1 applies to preflows as well.) Excesses are nonnegative for all vertices in $V - \{s\}$; because we have assumed that $U \subseteq V - \{s\}$, we must therefore have $e(v) = 0$ for all vertices $v \in U$. In particular, $e(u) = 0$, which contradicts the assumption that u is overflowing. ■

The next lemma bounds the heights of vertices, and its corollary bounds the number of relabel operations that are performed in total.

Lemma 26.21
Let $G = (V, E)$ be a flow network with source s and sink t. At any time during the execution of GENERIC-PUSH-RELABEL on G, we have $h[u] \le 2|V| - 1$ for all vertices $u \in V$.

Proof The heights of the source s and the sink t never change because these vertices are by definition not overflowing. Thus, we always have $h[s] = |V|$ and $h[t] = 0$, both of which are no greater than $2|V| - 1$.

Now consider any vertex $u \in V - \{s, t\}$. Initially, $h[u] = 0 \le 2|V| - 1$. We shall show that after each relabeling operation, we still have $h[u] \le 2|V| - 1$. When u is relabeled, it is overflowing, and Lemma 26.20 tells us that there is a simple path p from u to s in G_f. Let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = u$, $v_k = s$, and $k \le |V| - 1$ because p is simple. For $i = 0, 1, \ldots, k - 1$, we have $(v_i, v_{i+1}) \in E_f$, and therefore, by Lemma 26.17, $h[v_i] \le h[v_{i+1}] + 1$. Expanding these inequalities over path p yields $h[u] = h[v_0] \le h[v_k] + k \le h[s] + (|V| - 1) = 2|V| - 1$. ■

Corollary 26.22 (Bound on relabel operations)
Let $G = (V, E)$ be a flow network with source s and sink t. Then, during the execution of GENERIC-PUSH-RELABEL on G, the number of relabel operations is at most $2|V| - 1$ per vertex and at most $(2|V| - 1)(|V| - 2) < 2|V|^2$ overall.

Proof Only the $|V|-2$ vertices in $V-\{s,t\}$ may be relabeled. Let $u \in V-\{s,t\}$. The operation RELABEL(u) increases $h[u]$. The value of $h[u]$ is initially 0 and by Lemma 26.21 grows to at most $2|V|-1$. Thus, each vertex $u \in V-\{s,t\}$ is relabeled at most $2|V|-1$ times, and the total number of relabel operations performed is at most $(2|V|-1)(|V|-2) < 2|V|^2$. ■

Lemma 26.21 also helps us to bound the number of saturating pushes.

Lemma 26.23 (Bound on saturating pushes)
During the execution of GENERIC-PUSH-RELABEL on any flow network $G = (V, E)$, the number of saturating pushes is less than $2|V||E|$.

Proof For any pair of vertices $u, v \in V$, we will count the saturating pushes from u to v and from v to u together, calling them the saturating pushes between u and v. If there are any such pushes, at least one of (u,v) and (v,u) is actually an edge in E. Now, suppose that a saturating push from u to v has occurred. At that time, $h[v] = h[u]-1$. In order for another push from u to v to occur later, the algorithm must first push flow from v to u, which cannot happen until $h[v] = h[u]+1$. Since $h[u]$ never decreases, in order for $h[v] = h[u]+1$, the value of $h[v]$ must increase by at least 2. Likewise, $h[u]$ must increase by at least 2 between saturating pushes from v to u. Heights start at 0 and, by Lemma 26.21, never exceed $2|V|-1$, which implies that the number of times any vertex can have its height increase by 2 is less than $|V|$. Since at least one of $h[u]$ and $h[v]$ must increase by 2 between any two saturating pushes between u and v, there are fewer than $2|V|$ saturating pushes between u and v. Multiplying by the number of edges gives a bound of less than $2|V||E|$ on the total number of saturating pushes. ■

The following lemma bounds the number of nonsaturating pushes in the generic push-relabel algorithm.

Lemma 26.24 (Bound on nonsaturating pushes)
During the execution of GENERIC-PUSH-RELABEL on any flow network $G = (V, E)$, the number of nonsaturating pushes is less than $4|V|^2(|V|+|E|)$.

Proof Define a potential function $\Phi = \sum_{v:e(v)>0} h[v]$. Initially, $\Phi = 0$, and the value of Φ may change after each relabeling, saturating push, and nonsaturating push. We will bound the amount that saturating pushes and relabelings can contribute to the increase of Φ. Then we will show that each nonsaturating push must decrease Φ by at least 1, and will use these bounds to derive an upper bound on the number of nonsaturating pushes.

Let us examine the two ways in which Φ might increase. First, relabeling a vertex u increases Φ by less than $2|V|$, since the set over which the sum is taken is the same and the relabeling cannot increase u's height by more than its maximum possible height, which, by Lemma 26.21, is at most $2|V|-1$. Second, a saturating push from a vertex u to a vertex v increases Φ by less than $2|V|$, since no heights change and only vertex v, whose height is at most $2|V|-1$, can possibly become overflowing.

Now we show that a nonsaturating push from u to v decreases Φ by at least 1. Why? Before the nonsaturating push, u was overflowing, and v may or may not have been overflowing. By Lemma 26.14, u is no longer overflowing after the push. In addition, unless v is the source, it may or may not be overflowing after the push. Therefore, the potential function Φ has decreased by exactly $h[u]$, and it has increased by either 0 or $h[v]$. Since $h[u]-h[v]=1$, the net effect is that the potential function has decreased by at least 1.

Thus, during the course of the algorithm, the total amount of increase in Φ is due to relabelings and saturated pushes and is constrained by Corollary 26.22 and Lemma 26.23 to be less than $(2|V|)(2|V|^2)+(2|V|)(2|V||E|)=4|V|^2(|V|+|E|)$. Since $\Phi \geq 0$, the total amount of decrease, and therefore the total number of nonsaturating pushes, is less than $4|V|^2(|V|+|E|)$. ■

Having bounded the number of relabelings, saturating pushes, and nonsaturating push, we have set the stage for the following analysis of the GENERIC-PUSH-RELABEL procedure, and hence of any algorithm based on the push-relabel method.

Theorem 26.25
During the execution of GENERIC-PUSH-RELABEL on any flow network $G=(V,E)$, the number of basic operations is $O(V^2E)$.

Proof Immediate from Corollary 26.22 and Lemmas 26.23 and 26.24. ■

Thus, the algorithm terminates after $O(V^2E)$ operations. All that remains is to give an efficient method for implementing each operation and for choosing an appropriate operation to execute.

Corollary 26.26
There is an implementation of the generic push-relabel algorithm that runs in $O(V^2E)$ time on any flow network $G=(V,E)$.

Proof Exercise 26.4-1 asks you to show how to implement the generic algorithm with an overhead of $O(V)$ per relabel operation and $O(1)$ per push. It also asks you to design a data structure that allows you to pick an applicable operation in $O(1)$ time. The corollary then follows. ■

Exercises

26.4-1
Show how to implement the generic push-relabel algorithm using $O(V)$ time per relabel operation, $O(1)$ time per push, and $O(1)$ time to select an applicable operation, for a total time of $O(V^2E)$.

26.4-2
Prove that the generic push-relabel algorithm spends a total of only $O(VE)$ time in performing all the $O(V^2)$ relabel operations.

26.4-3
Suppose that a maximum flow has been found in a flow network $G = (V, E)$ using a push-relabel algorithm. Give a fast algorithm to find a minimum cut in G.

26.4-4
Give an efficient push-relabel algorithm to find a maximum matching in a bipartite graph. Analyze your algorithm.

26.4-5
Suppose that all edge capacities in a flow network $G = (V, E)$ are in the set $\{1, 2, \ldots, k\}$. Analyze the running time of the generic push-relabel algorithm in terms of $|V|$, $|E|$, and k. (*Hint:* How many times can each edge support a nonsaturating push before it becomes saturated?)

26.4-6
Show that line 7 of INITIALIZE-PREFLOW can be changed to

```
7   $h[s] \leftarrow |V[G]| - 2$
```

without affecting the correctness or asymptotic performance of the generic push-relabel algorithm.

26.4-7
Let $\delta_f(u, v)$ be the distance (number of edges) from u to v in the residual network G_f. Show that GENERIC-PUSH-RELABEL maintains the properties that $h[u] < |V|$ implies $h[u] \leq \delta_f(u, t)$ and that $h[u] \geq |V|$ implies $h[u] - |V| \leq \delta_f(u, s)$.

26.4-8 ⋆
As in the previous exercise, let $\delta_f(u, v)$ be the distance from u to v in the residual network G_f. Show how the generic push-relabel algorithm can be modified to maintain the property that $h[u] < |V|$ implies $h[u] = \delta_f(u, t)$ and that $h[u] \geq |V|$

implies $h[u] - |V| = \delta_f(u, s)$. The total time that your implementation dedicates to maintaining this property should be $O(VE)$.

26.4-9
Show that the number of nonsaturating pushes executed by GENERIC-PUSH-RELABEL on a flow network $G = (V, E)$ is at most $4|V|^2|E|$ for $|V| \geq 4$.

⋆ 26.5 The relabel-to-front algorithm

The push-relabel method allows us to apply the basic operations in any order at all. By choosing the order carefully and managing the network data structure efficiently, however, we can solve the maximum-flow problem faster than the $O(V^2E)$ bound given by Corollary 26.26. We shall now examine the relabel-to-front algorithm, a push-relabel algorithm whose running time is $O(V^3)$, which is asymptotically at least as good as $O(V^2E)$, and better for dense networks.

The relabel-to-front algorithm maintains a list of the vertices in the network. Beginning at the front, the algorithm scans the list, repeatedly selecting an overflowing vertex u and then "discharging" it, that is, performing push and relabel operations until u no longer has a positive excess. Whenever a vertex is relabeled, it is moved to the front of the list (hence the name "relabel-to-front") and the algorithm begins its scan anew.

The correctness and analysis of the relabel-to-front algorithm depend on the notion of "admissible" edges: those edges in the residual network through which flow can be pushed. After proving some properties about the network of admissible edges, we shall investigate the discharge operation and then present and analyze the relabel-to-front algorithm itself.

Admissible edges and networks

If $G = (V, E)$ is a flow network with source s and sink t, f is a preflow in G, and h is a height function, then we say that (u, v) is an ***admissible edge*** if $c_f(u, v) > 0$ and $h(u) = h(v) + 1$. Otherwise, (u, v) is ***inadmissible***. The ***admissible network*** is $G_{f,h} = (V, E_{f,h})$, where $E_{f,h}$ is the set of admissible edges.

The admissible network consists of those edges through which flow can be pushed. The following lemma shows that this network is a directed acyclic graph (dag).

Lemma 26.27 (The admissible network is acyclic)
If $G = (V, E)$ is a flow network, f is a preflow in G, and h is a height function on G, then the admissible network $G_{f,h} = (V, E_{f,h})$ is acyclic.

Proof The proof is by contradiction. Suppose that $G_{f,h}$ contains a cycle $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$ and $k > 0$. Since each edge in p is admissible, we have $h(v_{i-1}) = h(v_i) + 1$ for $i = 1, 2, \ldots, k$. Summing around the cycle gives

$$\begin{aligned} \sum_{i=1}^{k} h(v_{i-1}) &= \sum_{i=1}^{k} (h(v_i) + 1) \\ &= \sum_{i=1}^{k} h(v_i) + k \,. \end{aligned}$$

Because each vertex in cycle p appears once in each of the summations, we derive the contradiction that $0 = k$. ■

The next two lemmas show how push and relabel operations change the admissible network.

Lemma 26.28
Let $G = (V, E)$ be a flow network, let f be a preflow in G, and suppose that the attribute h is a height function. If a vertex u is overflowing and (u, v) is an admissible edge, then PUSH(u, v) applies. The operation does not create any new admissible edges, but it may cause (u, v) to become inadmissible.

Proof By the definition of an admissible edge, flow can be pushed from u to v. Since u is overflowing, the operation PUSH(u, v) applies. The only new residual edge that can be created by pushing flow from u to v is the edge (v, u). Since $h[v] = h[u] - 1$, edge (v, u) cannot become admissible. If the operation is a saturating push, then $c_f(u, v) = 0$ afterward and (u, v) becomes inadmissible. ■

Lemma 26.29
Let $G = (V, E)$ be a flow network, let f be a preflow in G, and suppose that the attribute h is a height function. If a vertex u is overflowing and there are no admissible edges leaving u, then RELABEL(u) applies. After the relabel operation, there is at least one admissible edge leaving u, but there are no admissible edges entering u.

Proof If u is overflowing, then by Lemma 26.15, either a push or a relabel operation applies to it. If there are no admissible edges leaving u, then no flow can be pushed from u and so RELABEL(u) applies. After the relabel operation, $h[u] = 1 + \min\{h[v] : (u, v) \in E_f\}$. Thus, if v is a vertex that realizes the minimum in this set, the edge (u, v) becomes admissible. Hence, after the relabel, there is at least one admissible edge leaving u.

To show that no admissible edges enter u after a relabel operation, suppose that there is a vertex v such that (v, u) is admissible. Then, $h[v] = h[u] + 1$ after the relabel, and so $h[v] > h[u] + 1$ just before the relabel. But by Lemma 26.13, no residual edges exist between vertices whose heights differ by more than 1. Moreover, relabeling a vertex does not change the residual network. Thus, (v, u) is not in the residual network, and hence it cannot be in the admissible network. ■

Neighbor lists

Edges in the relabel-to-front algorithm are organized into "neighbor lists." Given a flow network $G = (V, E)$, the ***neighbor list*** $N[u]$ for a vertex $u \in V$ is a singly linked list of the neighbors of u in G. Thus, vertex v appears in the list $N[u]$ if $(u, v) \in E$ or $(v, u) \in E$. The neighbor list $N[u]$ contains exactly those vertices v for which there may be a residual edge (u, v). The first vertex in $N[u]$ is pointed to by *head*$[N[u]]$. The vertex following v in a neighbor list is pointed to by *next-neighbor*$[v]$; this pointer is NIL if v is the last vertex in the neighbor list.

The relabel-to-front algorithm cycles through each neighbor list in an arbitrary order that is fixed throughout the execution of the algorithm. For each vertex u, the field *current*$[u]$ points to the vertex currently under consideration in $N[u]$. Initially, *current*$[u]$ is set to *head*$[N[u]]$.

Discharging an overflowing vertex

An overflowing vertex u is ***discharged*** by pushing all of its excess flow through admissible edges to neighboring vertices, relabeling u as necessary to cause edges leaving u to become admissible. The pseudocode goes as follows.

```
DISCHARGE($u$)
1  while $e[u] > 0$
2      do $v \leftarrow current[u]$
3         if $v = $ NIL
4           then RELABEL($u$)
5                $current[u] \leftarrow head[N[u]]$
6         elseif $c_f(u, v) > 0$ and $h[u] = h[v] + 1$
7           then PUSH($u, v$)
8         else $current[u] \leftarrow next\text{-}neighbor[v]$
```

Figure 26.9 steps through several iterations of the **while** loop of lines 1–8, which executes as long as vertex u has positive excess. Each iteration performs exactly one of three actions, depending on the current vertex v in the neighbor list $N[u]$.

1. If v is NIL, then we have run off the end of $N[u]$. Line 4 relabels vertex u, and then line 5 resets the current neighbor of u to be the first one in $N[u]$. (Lemma 26.30 below states that the relabel operation applies in this situation.)
2. If v is non-NIL and (u, v) is an admissible edge (determined by the test in line 6), then line 7 pushes some (or possibly all) of u's excess to vertex v.

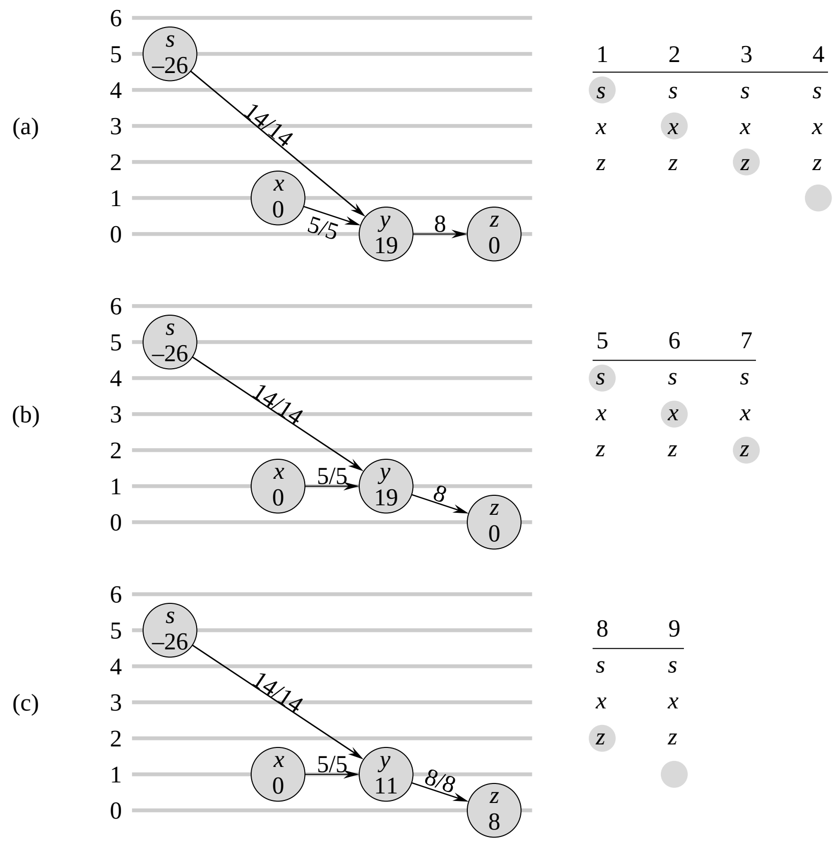
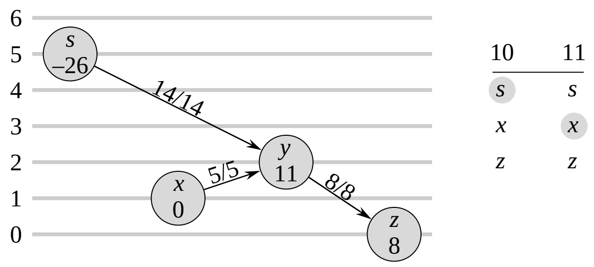
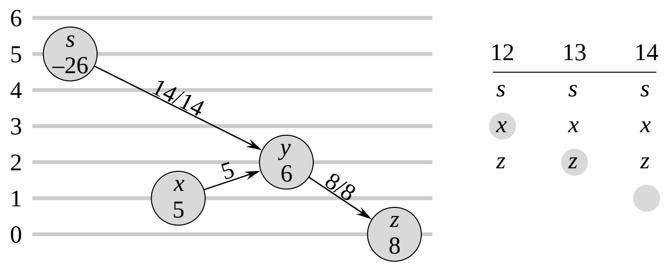


Figure 26.9 Discharging a vertex y. It takes 15 iterations of the **while** loop of DISCHARGE to push all the excess flow from y. Only the neighbors of y and edges entering or leaving y are shown. In each part, the number inside each vertex is its excess at the beginning of the first iteration shown in the part, and each vertex is shown at its height throughout the part. To the right is shown the neighbor list $N[y]$ at the beginning of each iteration, with the iteration number on top. The shaded neighbor is *current*$[y]$. **(a)** Initially, there are 19 units of excess to push from y, and *current*$[y] = s$. Iterations 1, 2, and 3 just advance *current*$[y]$, since there are no admissible edges leaving y. In iteration 4, *current*$[y] =$ NIL (shown by the shading being below the neighbor list), and so y is relabeled and *current*$[y]$ is reset to the head of the neighbor list. **(b)** After relabeling, vertex y has height 1. In iterations 5 and 6, edges (y, s) and (y, x) are found to be inadmissible, but 8 units of excess flow are pushed from y to z in iteration 7. Because of the push, *current*$[y]$ is not advanced in this iteration. **(c)** Because the push in iteration 7 saturated edge (y, z), it is found inadmissible in iteration 8. In iteration 9, *current*$[y] =$ NIL, and so vertex y is again relabeled and *current*$[y]$ is reset. **(d)** In iteration 10, (y, s) is inadmissible, but 5 units of excess flow are pushed from y to x in iteration 11. **(e)** Because *current*$[y]$ was not advanced in iteration 11, iteration 12 finds (y, x) to be inadmissible. Iteration 13 finds (y, z) inadmissible, and iteration 14 relabels vertex y and resets *current*$[y]$. **(f)** Iteration 15 pushes 6 units of excess flow from y to s. **(g)** Vertex y now has no excess flow, and DISCHARGE terminates. In this example, DISCHARGE both starts and finishes with the current pointer at the head of the neighbor list, but in general this need not be the case.

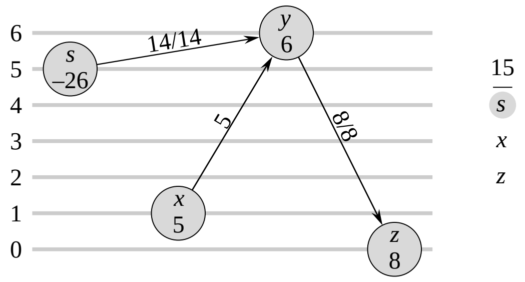
(d)


6
5
4
3
2
1
0
s
–26
14/14
y
11
5/5
x
0
8/8
z
8
10
11
s
s
x
x
z
z

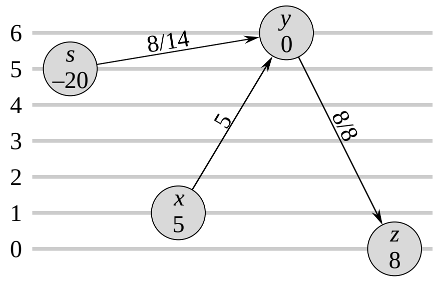
(e)


6
5
4
3
2
1
0
s
–26
14/14
y
6
5
x
5
8/8
z
8
12
13
14
s
s
s
x
x
x
z
z
z

(f)


6
5
4
3
2
1
0
s
–26
14/14
y
6
5
8/8
x
5
z
8
15
s
x
z

(g)


6
5
4
3
2
1
0
s
–20
8/14
y
0
5
8/8
x
5
z
8

3. If v is non-NIL but (u, v) is inadmissible, then line 8 advances *current*$[u]$ one position further in the neighbor list $N[u]$.

Observe that if DISCHARGE is called on an overflowing vertex u, then the last action performed by DISCHARGE must be a push from u. Why? The procedure terminates only when $e[u]$ becomes zero, and neither the relabel operation nor the advancing of the pointer *current*$[u]$ affects the value of $e[u]$.

We must be sure that when PUSH or RELABEL is called by DISCHARGE, the operation applies. The next lemma proves this fact.

Lemma 26.30
If DISCHARGE calls PUSH(u, v) in line 7, then a push operation applies to (u, v). If DISCHARGE calls RELABEL(u) in line 4, then a relabel operation applies to u.

Proof The tests in lines 1 and 6 ensure that a push operation occurs only if the operation applies, which proves the first statement in the lemma.

To prove the second statement, according to the test in line 1 and Lemma 26.29, we need only show that all edges leaving u are inadmissible. Observe that as DISCHARGE(u) is repeatedly called, the pointer *current*$[u]$ moves down the list $N[u]$. Each "pass" begins at the head of $N[u]$ and finishes with *current*$[u] =$ NIL, at which point u is relabeled and a new pass begins. For the *current*$[u]$ pointer to advance past a vertex $v \in N[u]$ during a pass, the edge (u, v) must be deemed inadmissible by the test in line 6. Thus, by the time the pass completes, every edge leaving u has been determined to be inadmissible at some time during the pass. The key observation is that at the end of the pass, every edge leaving u is still inadmissible. Why? By Lemma 26.28, pushes cannot create any admissible edges, let alone one leaving u. Thus, any admissible edge must be created by a relabel operation. But the vertex u is not relabeled during the pass, and by Lemma 26.29, any other vertex v that is relabeled during the pass has no entering admissible edges after relabeling. Thus, at the end of the pass, all edges leaving u remain inadmissible, and the lemma is proved. ■

The relabel-to-front algorithm

In the relabel-to-front algorithm, we maintain a linked list L consisting of all vertices in $V - \{s, t\}$. A key property is that the vertices in L are topologically sorted according to the admissible network, as we shall see in the loop invariant below. (Recall from Lemma 26.27 that the admissible network is a dag.)

The pseudocode for the relabel-to-front algorithm assumes that the neighbor lists $N[u]$ have already been created for each vertex u. It also assumes that *next*$[u]$ points to the vertex that follows u in list L and that, as usual, *next*$[u] =$ NIL if u is the last vertex in the list.

```
RELABEL-TO-FRONT$(G, s, t)$
 1  INITIALIZE-PREFLOW$(G, s)$
 2  $L \leftarrow V[G] - \{s, t\}$, in any order
 3  for each vertex $u \in V[G] - \{s, t\}$
 4       do $current[u] \leftarrow head[N[u]]$
 5  $u \leftarrow head[L]$
 6  while $u \neq$ NIL
 7       do $old\text{-}height \leftarrow h[u]$
 8          DISCHARGE$(u)$
 9          if $h[u] > old\text{-}height$
10             then move $u$ to the front of list $L$
11          $u \leftarrow next[u]$
```

The relabel-to-front algorithm works as follows. Line 1 initializes the preflow and heights to the same values as in the generic push-relabel algorithm. Line 2 initializes the list L to contain all potentially overflowing vertices, in any order. Lines 3–4 initialize the *current* pointer of each vertex u to the first vertex in u's neighbor list.

As shown in Figure 26.10, the **while** loop of lines 6–11 runs through the list L, discharging vertices. Line 5 makes it start with the first vertex in the list. Each time through the loop, a vertex u is discharged in line 8. If u was relabeled by the DISCHARGE procedure, line 10 moves it to the front of list L. This determination is made by saving u's height in the variable *old-height* before the discharge operation (line 7) and comparing this saved height to u's height afterward (line 9). Line 11 makes the next iteration of the **while** loop use the vertex following u in list L. If u was moved to the front of the list, the vertex used in the next iteration is the one following u in its new position in the list.

To show that RELABEL-TO-FRONT computes a maximum flow, we shall show that it is an implementation of the generic push-relabel algorithm. First, observe that it performs push and relabel operation only when they apply, since Lemma 26.30 guarantees that DISCHARGE only performs them when they apply. It remains to show that when RELABEL-TO-FRONT terminates, no basic operations apply. The remainder of the correctness argument relies on the following loop invariant:

> At each test in line 6 of RELABEL-TO-FRONT, list L is a topological sort of the vertices in the admissible network $G_{f,h} = (V, E_{f,h})$, and no vertex before u in the list has excess flow.

Initialization: Immediately after INITIALIZE-PREFLOW has been run, $h[s] = |V|$ and $h[v] = 0$ for all $v \in V - \{s\}$. Since $|V| \geq 2$ (because V contains at least s and t), no edge can be admissible. Thus, $E_{f,h} = \emptyset$, and any ordering of $V - \{s, t\}$ is a topological sort of $G_{f,h}$.

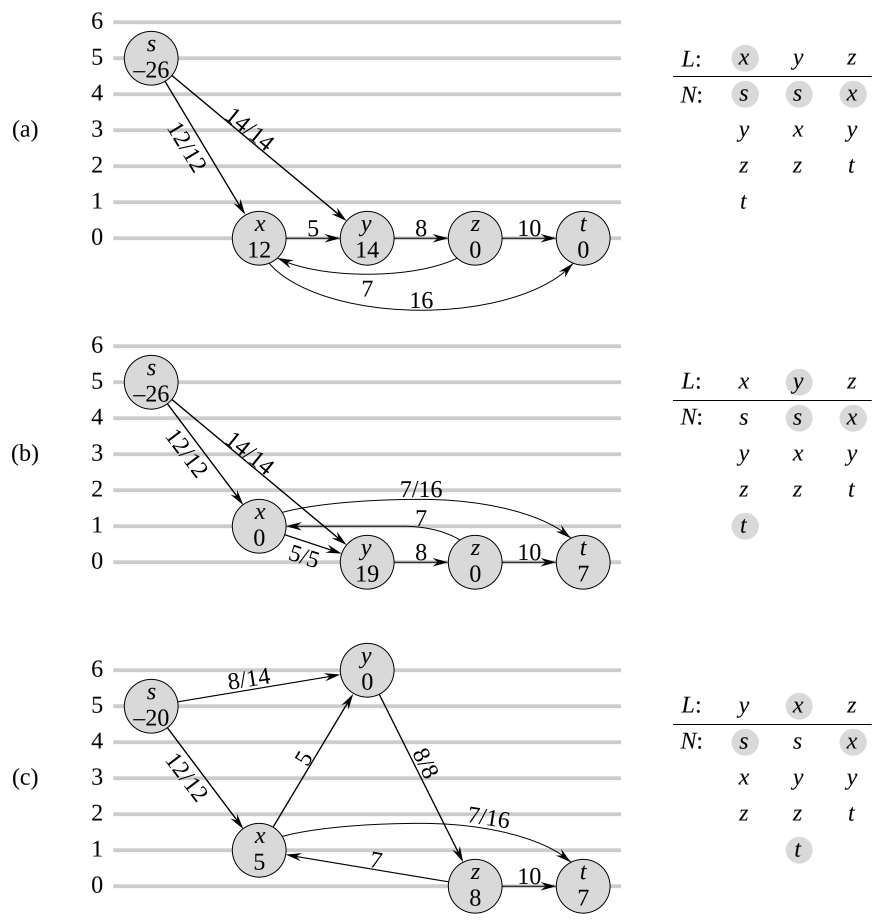
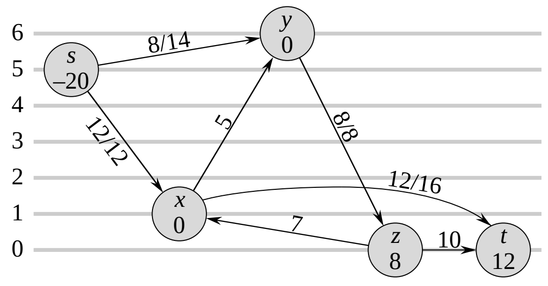


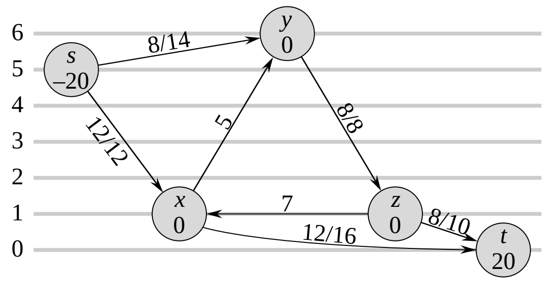
Figure 26.10 The action of RELABEL-TO-FRONT. **(a)** A flow network just before the first iteration of the **while** loop. Initially, 26 units of flow leave source s. On the right is shown the initial list $L = \langle x, y, z \rangle$, where initially $u = x$. Under each vertex in list L is its neighbor list, with the current neighbor shaded. Vertex x is discharged. It is relabeled to height 1, 5 units of excess flow are pushed to y, and the 7 remaining units of excess are pushed to the sink t. Because x is relabeled, it is moved to the head of L, which in this case does not change the structure of L. **(b)** After x, the next vertex in L that is discharged is y. Figure 26.9 shows the detailed action of discharging y in this situation. Because y is relabeled, it is moved to the head of L. **(c)** Vertex x now follows y in L, and so it is again discharged, pushing all 5 units of excess flow to t. Because vertex x is not relabeled in this discharge operation, it remains in place in list L. **(d)** Since vertex z follows vertex x in L, it is discharged. It is relabeled to height 1 and all 8 units of excess flow are pushed to t. Because z is relabeled, it is moved to the front of L. **(e)** Vertex y now follows vertex z in L and is therefore discharged. But because y has no excess, DISCHARGE immediately returns, and y remains in place in L. Vertex x is then discharged. Because it, too, has no excess, DISCHARGE again returns, and x remains in place in L. RELABEL-TO-FRONT has reached the end of list L and terminates. There are no overflowing vertices, and the preflow is a maximum flow.

(d)



L:	y	x	z
N:	s	s	x
	x	y	y
	z	z	t
		t	

(e)



L:	z	y	x
N:	x	s	s
	y	x	y
	t	z	z
			t

Since u is initially the head of the list L, there are no vertices before it and so there are none before it with excess flow.

Maintenance: To see that the topological sort is maintained by each iteration of the **while** loop, we start by observing that the admissible network is changed only by push and relabel operations. By Lemma 26.28, push operations do not cause edges to become admissible. Thus, admissible edges can be created only by relabel operations. After a vertex u is relabeled, however, Lemma 26.29 states that there are no admissible edges entering u but there may be admissible edges leaving u. Thus, by moving u to the front of L, the algorithm ensures that any admissible edges leaving u satisfy the topological sort ordering.

To see that no vertex preceding u in L has excess flow, we denote the vertex that will be u in the next iteration by u'. The vertices that will precede u' in the next iteration include the current u (due to line 11) and either no other vertices (if u is relabeled) or the same vertices as before (if u is not relabeled). Since u is discharged, it has no excess flow afterward. Thus, if u is relabeled during the discharge, no vertices preceding u' have excess flow. If u is not relabeled during the discharge, no vertices before it on the list acquired excess flow during this discharge, because L remained topologically sorted at all times during the discharge (as pointed out just above, admissible edges are created only by relabeling, not pushing), and so each push operation causes excess flow to move only to vertices further down the list (or to s or t). Again, no vertices preceding u' have excess flow.

Termination: When the loop terminates, u is just past the end of L, and so the loop invariant ensures that the excess of every vertex is 0. Thus, no basic operations apply.

Analysis

We shall now show that RELABEL-TO-FRONT runs in $O(V^3)$ time on any flow network $G = (V, E)$. Since the algorithm is an implementation of the generic push-relabel algorithm, we shall take advantage of Corollary 26.22, which provides an $O(V)$ bound on the number of relabel operations executed per vertex and an $O(V^2)$ bound on the total number of relabel operations overall. In addition, Exercise 26.4-2 provides an $O(VE)$ bound on the total time spent performing relabel operations, and Lemma 26.23 provides an $O(VE)$ bound on the total number of saturating push operations.

Theorem 26.31
The running time of RELABEL-TO-FRONT on any flow network $G = (V, E)$ is $O(V^3)$.

Proof Let us consider a "phase" of the relabel-to-front algorithm to be the time between two consecutive relabel operations. There are $O(V^2)$ phases, since there are $O(V^2)$ relabel operations. Each phase consists of at most $|V|$ calls to DISCHARGE, which can be seen as follows. If DISCHARGE does not perform a relabel operation, then the next call to DISCHARGE is further down the list L, and the length of L is less than $|V|$. If DISCHARGE does perform a relabel, the next call to DISCHARGE belongs to a different phase. Since each phase contains at most $|V|$ calls to DISCHARGE and there are $O(V^2)$ phases, the number of times DISCHARGE is called in line 8 of RELABEL-TO-FRONT is $O(V^3)$. Thus, the total work performed by the **while** loop in RELABEL-TO-FRONT, excluding the work performed within DISCHARGE, is at most $O(V^3)$.

We must now bound the work performed within DISCHARGE during the execution of the algorithm. Each iteration of the **while** loop within DISCHARGE performs one of three actions. We shall analyze the total amount of work involved in performing each of these actions.

We start with relabel operations (lines 4–5). Exercise 26.4-2 provides an $O(VE)$ time bound on all the $O(V^2)$ relabels that are performed.

Now, suppose that the action updates the *current*$[u]$ pointer in line 8. This action occurs $O(\text{degree}(u))$ times each time a vertex u is relabeled, and $O(V \cdot \text{degree}(u))$ times overall for the vertex. For all vertices, therefore, the total amount of work done in advancing pointers in neighbor lists is $O(VE)$ by the handshaking lemma (Exercise B.4-1).

The third type of action performed by DISCHARGE is a push operation (line 7). We already know that the total number of saturating push operations is $O(VE)$. Observe that if a nonsaturating push is executed, DISCHARGE immediately returns, since the push reduces the excess to 0. Thus, there can be at most one nonsaturating push per call to DISCHARGE. As we have observed, DISCHARGE is called $O(V^3)$ times, and thus the total time spent performing nonsaturating pushes is $O(V^3)$.

The running time of RELABEL-TO-FRONT is therefore $O(V^3 + VE)$, which is $O(V^3)$. ■

Exercises

26.5-1

Illustrate the execution of RELABEL-TO-FRONT in the manner of Figure 26.10 for the flow network in Figure 26.1(a). Assume that the initial ordering of vertices in L is $\langle v_1, v_2, v_3, v_4 \rangle$ and that the neighbor lists are

$$\begin{aligned} N[v_1] &= \langle s, v_2, v_3 \rangle , \\ N[v_2] &= \langle s, v_1, v_3, v_4 \rangle , \\ N[v_3] &= \langle v_1, v_2, v_4, t \rangle , \\ N[v_4] &= \langle v_2, v_3, t \rangle . \end{aligned}$$

26.5-2 ⋆

We would like to implement a push-relabel algorithm in which we maintain a first-in, first-out queue of overflowing vertices. The algorithm repeatedly discharges the vertex at the head of the queue, and any vertices that were not overflowing before the discharge but are overflowing afterward are placed at the end of the queue. After the vertex at the head of the queue is discharged, it is removed. When the queue is empty, the algorithm terminates. Show that this algorithm can be implemented to compute a maximum flow in $O(V^3)$ time.

26.5-3

Show that the generic algorithm still works if RELABEL updates $h[u]$ by simply computing $h[u] \leftarrow h[u] + 1$. How would this change affect the analysis of RELABEL-TO-FRONT?

26.5-4 ⋆

Show that if we always discharge a highest overflowing vertex, the push-relabel method can be made to run in $O(V^3)$ time.

26.5-5

Suppose that at some point in the execution of a push-relabel algorithm, there exists an integer $0 < k \leq |V| - 1$ for which no vertex has $h[v] = k$. Show that all vertices with $h[v] > k$ are on the source side of a minimum cut. If such a k exists, the ***gap heuristic*** updates every vertex $v \in V - s$ for which $h[v] > k$ to set $h[v] \leftarrow \max(h[v], |V|+1)$. Show that the resulting attribute h is a height function. (The gap heuristic is crucial in making implementations of the push-relabel method perform well in practice.)

Problems

26-1 Escape problem

An $n \times n$ ***grid*** is an undirected graph consisting of n rows and n columns of vertices, as shown in Figure 26.11. We denote the vertex in the ith row and the jth column by (i, j). All vertices in a grid have exactly four neighbors, except for the boundary vertices, which are the points (i, j) for which $i = 1$, $i = n$, $j = 1$, or $j = n$.

Given $m \leq n^2$ starting points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ in the grid, the ***escape problem*** is to determine whether or not there are m vertex-disjoint paths from the starting points to any m different points on the boundary. For example, the grid in Figure 26.11(a) has an escape, but the grid in Figure 26.11(b) does not.

a. Consider a flow network in which vertices, as well as edges, have capacities. That is, the total positive flow entering any given vertex is subject to a capacity constraint. Show that determining the maximum flow in a network with edge and vertex capacities can be reduced to an ordinary maximum-flow problem on a flow network of comparable size.

b. Describe an efficient algorithm to solve the escape problem, and analyze its running time.

26-2 Minimum path cover

A ***path cover*** of a directed graph $G = (V, E)$ is a set P of vertex-disjoint paths such that every vertex in V is included in exactly one path in P. Paths may start and end anywhere, and they may be of any length, including 0. A ***minimum path cover*** of G is a path cover containing the fewest possible paths.

a. Give an efficient algorithm to find a minimum path cover of a directed acyclic graph $G = (V, E)$. (*Hint:* Assuming that $V = \{1, 2, \ldots, n\}$, construct the graph $G' = (V', E')$, where

$$\begin{aligned} V' &= \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\} , \\ E' &= \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\} , \end{aligned}$$

and run a maximum-flow algorithm.)

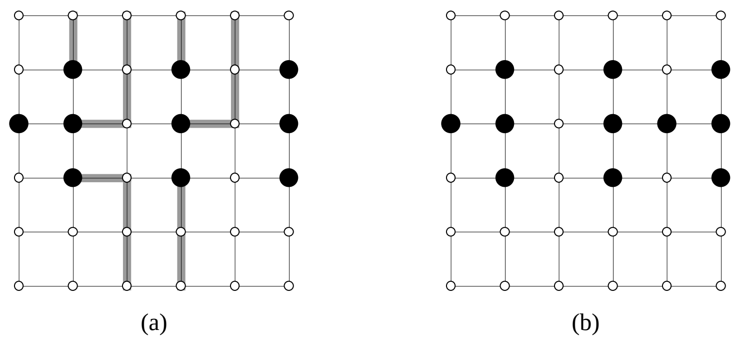


Figure 26.11 Grids for the escape problem. Starting points are black, and other grid vertices are white. **(a)** A grid with an escape, shown by shaded paths. **(b)** A grid with no escape.

b. Does your algorithm work for directed graphs that contain cycles? Explain.

26-3 Space shuttle experiments

Professor Spock is consulting for NASA, which is planning a series of space shuttle flights and must decide which commercial experiments to perform and which instruments to have on board each flight. For each flight, NASA considers a set $E = \{E_1, E_2, \ldots, E_m\}$ of experiments, and the commercial sponsor of experiment E_j has agreed to pay NASA p_j dollars for the results of the experiment. The experiments use a set $I = \{I_1, I_2, \ldots, I_n\}$ of instruments; each experiment E_j requires all the instruments in a subset $R_j \subseteq I$. The cost of carrying instrument I_k is c_k dollars. The professor's job is to find an efficient algorithm to determine which experiments to perform and which instruments to carry for a given flight in order to maximize the net revenue, which is the total income from experiments performed minus the total cost of all instruments carried.

Consider the following network G. The network contains a source vertex s, vertices $I_1, I_2, \ldots, I_n$, vertices $E_1, E_2, \ldots, E_m$, and a sink vertex t. For $k = 1, 2 \ldots, n$, there is an edge (s, I_k) of capacity c_k, and for $j = 1, 2, \ldots, m$, there is an edge (E_j, t) of capacity p_j. For $k = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$, if $I_k \in R_j$, then there is an edge (I_k, E_j) of infinite capacity.

a. Show that if $E_j \in T$ for a finite-capacity cut (S, T) of G, then $I_k \in T$ for each $I_k \in R_j$.

b. Show how to determine the maximum net revenue from the capacity of the minimum cut of G and the given p_j values.

c. Give an efficient algorithm to determine which experiments to perform and which instruments to carry. Analyze the running time of your algorithm in terms of m, n, and $r = \sum_{j=1}^{m} |R_j|$.

26-4 Updating maximum flow

Let $G = (V, E)$ be a flow network with source s, sink t, and integer capacities. Suppose that we are given a maximum flow in G.

a. Suppose that the capacity of a single edge $(u, v) \in E$ is increased by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

b. Suppose that the capacity of a single edge $(u, v) \in E$ is decreased by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

26-5 Maximum flow by scaling

Let $G = (V, E)$ be a flow network with source s, sink t, and an integer capacity $c(u, v)$ on each edge $(u, v) \in E$. Let $C = \max_{(u,v)\in E} c(u, v)$.

a. Argue that a minimum cut of G has capacity at most $C\,|E|$.

b. For a given number K, show that an augmenting path of capacity at least K can be found in $O(E)$ time, if such a path exists.

The following modification of FORD-FULKERSON-METHOD can be used to compute a maximum flow in G.

```
MAX-FLOW-BY-SCALING(G, s, t)
1  $C \leftarrow \max_{(u,v)\in E} c(u, v)$
2  initialize flow $f$ to 0
3  $K \leftarrow 2^{\lfloor \lg C \rfloor}$
4  **while** $K \geq 1$
5        **do while** there exists an augmenting path $p$ of capacity at least $K$
6                  **do** augment flow $f$ along $p$
7             $K \leftarrow K/2$
8  **return** $f$
```

c. Argue that MAX-FLOW-BY-SCALING returns a maximum flow.

d. Show that the capacity of a minimum cut of the residual graph G_f is at most $2K\,|E|$ each time line 4 is executed.

e. Argue that the inner **while** loop of lines 5–6 is executed $O(E)$ times for each value of K.

f. Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs in $O(E^2 \lg C)$ time.

26-6 Maximum flow with negative capacities

Suppose that we allow a flow network to have negative (as well as positive) edge capacities. In such a network, a feasible flow need not exist.

a. Consider an edge (u, v) in a flow network $G = (V, E)$ with $c(u, v) < 0$. Briefly explain what such a negative capacity means in terms of the flow between u and v.

Let $G = (V, E)$ be a flow network with negative edge capacities, and let s and t be the source and sink of G. Construct the ordinary flow network $G' = (V', E')$ with capacity function c', source s', and sink t', where

$$V' = V \cup \{s', t'\}$$

and

$$\begin{aligned} E' = E &\cup \{(u, v) : (v, u) \in E\} \\ &\cup \{(s', v) : v \in V\} \\ &\cup \{(u, t') : u \in V\} \\ &\cup \{(s, t), (t, s)\} \ . \end{aligned}$$

We assign capacities to edges as follows. For each edge $(u, v) \in E$, we set

$$c'(u, v) = c'(v, u) = (c(u, v) + c(v, u))/2 \ .$$

For each vertex $u \in V$, we set

$$c'(s', u) = \max(0, (c(V, u) - c(u, V))/2)$$

and

$$c'(u, t') = \max(0, (c(u, V) - c(V, u))/2) \ .$$

We also set $c'(s, t) = c'(t, s) = \infty$.

b. Prove that if a feasible flow exists in G, then all capacities in G' are nonnegative and a maximum flow exists in G' such that all edges into the sink t' are saturated.

c. Prove the converse of part (b). Your proof should be constructive, that is, given a flow in G' that saturates all the edges into t', your proof should show how to obtain a feasible flow in G.

d. Describe an algorithm that finds a maximum feasible flow in G. Denote by $MF(|V|, |E|)$ the worst-case running time of an ordinary maximum flow algorithm on a graph with $|V|$ vertices and $|E|$ edges. Analyze your algorithm for computing the maximum flow of a flow network with negative capacities in terms of MF.

26-7 The Hopcroft-Karp bipartite matching algorithm

In this problem, we describe a faster algorithm, due to Hopcroft and Karp, for finding a maximum matching in a bipartite graph. The algorithm runs in $O(\sqrt{V}E)$ time. Given an undirected, bipartite graph $G = (V, E)$, where $V = L \cup R$ and all edges have exactly one endpoint in L, let M be a matching in G. We say that a simple path P in G is an ***augmenting path*** with respect to M if it starts at an unmatched vertex in L, ends at an unmatched vertex in R, and its edges belong alternately to M and $E - M$. (This definition of an augmenting path is related to, but different from, an augmenting path in a flow network.) In this problem, we treat a path as a sequence of edges, rather than as a sequence of vertices. A shortest augmenting path with respect to a matching M is an augmenting path with a minimum number of edges.

Given two sets A and B, the ***symmetric difference*** $A \oplus B$ is defined as $(A - B) \cup (B - A)$, that is, the elements that are in exactly one of the two sets.

a. Show that if M is a matching and P is an augmenting path with respect to M, then the symmetric difference $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$. Show that if $P_1, P_2, \ldots, P_k$ are vertex-disjoint augmenting paths with respect to M, then the symmetric difference $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ is a matching with cardinality $|M| + k$.

The general structure of our algorithm is the following:

```
HOPCROFT-KARP(G)
1  $M \leftarrow \emptyset$
2  repeat
3        let $\mathcal{P} \leftarrow \{P_1, P_2, \ldots, P_k\}$ be a maximal set of vertex-disjoint
             shortest augmenting paths with respect to $M$
4        $M \leftarrow M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$
5    until $\mathcal{P} = \emptyset$
6  return $M$
```

The remainder of this problem asks you to analyze the number of iterations in the algorithm (that is, the number of iterations in the **repeat** loop) and to describe an implementation of line 3.

b. Given two matchings M and M^* in G, show that every vertex in the graph $G' = (V, M \oplus M^*)$ has degree at most 2. Conclude that G' is a disjoint union of simple paths or cycles. Argue that edges in each such simple path or cycle belong alternately to M or M^*. Prove that if $|M| \le |M^*|$, then $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M.

Let l be the length of a shortest augmenting path with respect to a matching M, and let $P_1, P_2, \ldots, P_k$ be a maximal set of vertex-disjoint augmenting paths of length l with respect to M. Let $M' = M \oplus (P_1 \cup \cdots \cup P_k)$, and suppose that P is a shortest augmenting path with respect to M'.

c. Show that if P is vertex-disjoint from $P_1, P_2, \ldots, P_k$, then P has more than l edges.

d. Now suppose that P is not vertex-disjoint from $P_1, P_2, \ldots, P_k$. Let A be the set of edges $(M \oplus M') \oplus P$. Show that $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ and that $|A| \ge (k+1)l$. Conclude that P has more than l edges.

e. Prove that if a shortest augmenting path with respect to M has l edges, the size of the maximum matching is at most $|M| + |V|/(l+1)$.

f. Show that the number of **repeat** loop iterations in the algorithm is at most $2\sqrt{V}$. (*Hint:* By how much can M grow after iteration number $\sqrt{V}$?)

g. Give an algorithm that runs in $O(E)$ time to find a maximal set of vertex-disjoint shortest augmenting paths $P_1, P_2, \ldots, P_k$ for a given matching M. Conclude that the total running time of HOPCROFT-KARP is $O(\sqrt{V}E)$.

Chapter notes

Ahuja, Magnanti, and Orlin [7], Even [87], Lawler [196], Papadimitriou and Steiglitz [237], and Tarjan [292] are good references for network flow and related algorithms. Goldberg, Tardos, and Tarjan [119] also provide a nice survey of algorithms for network-flow problems, and Schrijver [267] has written an interesting review of historical developments in the field of network flows.

The Ford-Fulkerson method is due to Ford and Fulkerson [93], who originated the formal study of many of the problems in the area of network flow, including the maximum-flow and bipartite-matching problems. Many early implementations of the Ford-Fulkerson method found augmenting paths using breadth-first search; Edmonds and Karp [86], and independently Dinic [76], proved that this strategy yields a polynomial-time algorithm. A related idea, that of using "blocking flows," was also first developed by Dinic [76]. Karzanov [176] first developed the idea of preflows. The push-relabel method is due to Goldberg [117] and Goldberg and Tarjan [121]. Goldberg and Tarjan gave an $O(V^3)$-time algorithm that uses a queue to maintain the set of overflowing vertices, as well as an algorithm that uses dynamic trees to achieve a running time of $O(VE \lg(V^2/E+2))$. Several other researchers have developed push-relabel maximum-flow algorithms. Ahuja and Orlin [9] and Ahuja, Orlin, and Tarjan [10] gave algorithms that used scaling. Cheriyan and Maheshwari [55] proposed pushing flow from the overflowing vertex of maximum height. Cheriyan and Hagerup [54] suggested randomly permuting the neighbor lists, and several researchers [14, 178, 241] developed clever derandomizations of this idea, leading to a sequence of faster algorithms. The algorithm of King, Rao, and Tarjan [178] is the fastest such algorithm and runs in $O(VE \log_{E/(V \lg V)} V)$ time.

The asymptotically fastest algorithm to date for the maximum-flow problem is due to Goldberg and Rao [120] and runs in time $O(\min(V^{2/3}, E^{1/2})$ $E \lg(V^2/E + 2) \lg C)$, where $C = \max_{(u,v)\in E} c(u,v)$. This algorithm does not use the push-relabel method but instead is based on finding blocking flows. All previous maximum-flow algorithms, including the ones in this chapter, use some notion of distance (the push-relabel algorithms use the analogous notion of height), with a length of 1 assigned implicitly to each edge. This new algorithm takes a different approach and assigns a length of 0 to high-capacity edges and a length of 1 to low-capacity edges. Informally, with respect to these lengths, shortest paths from the source to the sink tend have high capacity, which means that fewer iterations need be performed.

In practice, push-relabel algorithms currently dominate augmenting-path or linear-programming based algorithms for the maximum-flow problem. A study by Cherkassky and Goldberg [56] underscores the importance of using two heuristics when implementing a push-relabel algorithm. The first heuristic is to periodically perform a breadth-first search of the residual graph in order to obtain more accurate height values. The second heuristic is the gap heuristic, described in Exericse 26.5-5. They conclude that the best choice of push-relabel variants is the one that chooses to discharge the overflowing vertex with the maximum height.

The best algorithm to date for maximum bipartite matching, discovered by Hopcroft and Karp [152], runs in $O(\sqrt{V}E)$ time and is described in Problem 26-7. The book by Lovász and Plummer [207] is an excellent reference on matching problems.