

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 model many problems, including liquids flowing through pipes, parts through assembly lines, current through electrical networks, and information through communication networks.

We can think of each directed edge in a flow network 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 we can ship material 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, we can adapt the basic techniques used in maximum-flow algorithms 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, appears 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** $c(u, v) \geq 0$. We further require that if E contains an edge (u, v) , then there is no edge (v, u) in the reverse direction. (We shall see shortly how to work around this restriction.) If $(u, v) \notin E$, then for convenience we define $c(u, v) = 0$, and we disallow self-loops. We distinguish two vertices in a flow network: a **source** s and a **sink** t . For convenience, we assume that each vertex lies on some path from the source to the sink. That is, for each vertex $v \in V$, the flow network contains a path $s \rightsquigarrow v \rightsquigarrow t$. The graph is therefore connected and, since each vertex other than s has at least one entering edge, $|E| \geq |V| - 1$. 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 $f : V \times V \rightarrow \mathbb{R}$ that satisfies the following **two properties**:

Capacity constraint: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.

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

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

When $(u, v) \notin E$, there can be no flow from u to v , and $f(u, v) = 0$.

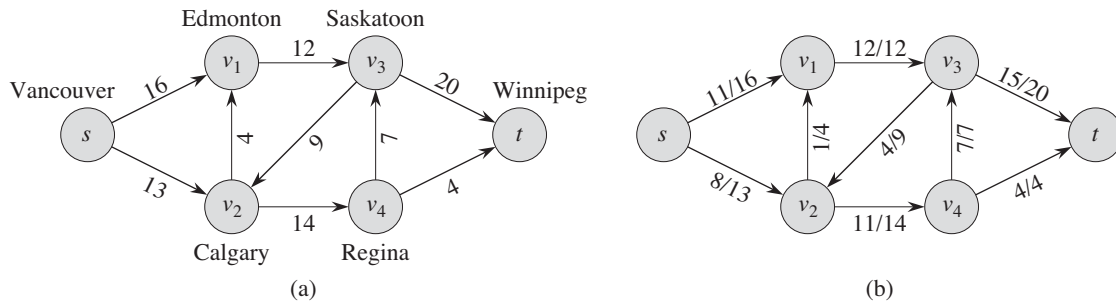


Figure 26.1 (a) A flow network $G = (V, E)$ for the Lucky Puck Company's trucking problem. The Vancouver factory is the source s , and the Winnipeg warehouse is the sink t . The company ships pucks through intermediate cities, but only $c(u, v)$ crates per day can go from city u to city v . Each edge is labeled with its capacity. (b) A flow f in G with value $|f| = 19$. Each edge (u, v) is labeled by $f(u, v)/c(u, v)$. The slash notation merely separates the flow and capacity; it does not indicate division.

We call the nonnegative quantity $f(u, v)$ the flow from vertex u to vertex v . The **value** $|f|$ of a flow f is defined as

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

that is, the total flow out of the source minus the flow into the source. (Here, the $|\cdot|$ notation denotes flow value, not absolute value or cardinality.) Typically, a flow network will not have any edges into the source, and the flow into the source, given by the summation $\sum_{v \in V} f(v, s)$, will be 0. We include it, however, because when we introduce residual networks later in this chapter, the flow into the source will become significant. In the **maximum-flow problem**, we are given a 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 definition of flow and the two flow properties. The capacity constraint simply says that the flow from one vertex to another must be nonnegative and must not exceed the given capacity. The flow-conservation property says that the total flow into a vertex other than the source or sink must equal the total flow out of that vertex—informally, “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

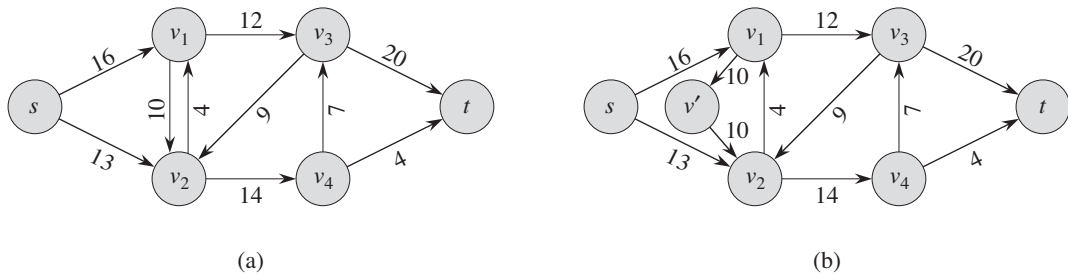


Figure 26.2 Converting a network with antiparallel edges to an equivalent one with no antiparallel edges. **(a)** A flow network containing both the edges (v_1, v_2) and (v_2, v_1) . **(b)** An equivalent network with no antiparallel edges. We add the new vertex v' , and we replace edge (v_1, v_2) by the pair of edges (v_1, v') and (v', v_2) , both with the same capacity as (v_1, v_2) .

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 the company cannot alter the flow network shown in Figure 26.1(a). They need to determine the largest number p of crates per day that they can ship 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.

We can 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, the model must obey flow conservation, 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.

Modeling problems with antiparallel edges

Suppose that the trucking firm offered Lucky Puck the opportunity to lease space for 10 crates in trucks going from Edmonton to Calgary. It would seem natural to add this opportunity to our example and form the network shown in Figure 26.2(a). This network suffers from one problem, however: it violates our original assumption that if an edge $(v_1, v_2) \in E$, then $(v_2, v_1) \notin E$. We call the two edges (v_1, v_2) and (v_2, v_1) **antiparallel**. Thus, if we wish to model a flow problem with antiparallel edges, we must transform the network into an equivalent one containing no

antiparallel edges. Figure 26.2(b) displays this equivalent network. We choose one of the two antiparallel edges, in this case (v_1, v_2) , and split it by adding a new vertex v' and replacing edge (v_1, v_2) with the pair of edges (v_1, v') and (v', v_2) . We also set the capacity of both new edges to the capacity of the original edge. The resulting network satisfies the property that if an edge is in the network, the reverse edge is not. Exercise 26.1-1 asks you to prove that the resulting network is equivalent to the original one.

Thus, we see that a real-world flow problem might be most naturally modeled by a network with antiparallel edges. It will be convenient to disallow antiparallel edges, however, and so we have a straightforward way to convert a network containing antiparallel edges into an equivalent one with no antiparallel edges.

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 $\{s_1, s_2, \dots, s_m\}$ and a set of n warehouses $\{t_1, t_2, \dots, t_n\}$, as shown in Figure 26.3(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 26.3(b) shows how to convert the network from (a) 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 $i = 1, 2, \dots, m$. We also create a new **supersink** t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, \dots, n$. 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 26.1-2 asks you to prove formally that the two problems are equivalent.

Exercises

26.1-1

Show that splitting an edge in a flow network yields an equivalent network. More formally, suppose that flow network G contains edge (u, v) , and we create a new flow network G' by creating a new vertex x and replacing (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$. Show that a maximum flow in G' has the same value as a maximum flow in G .

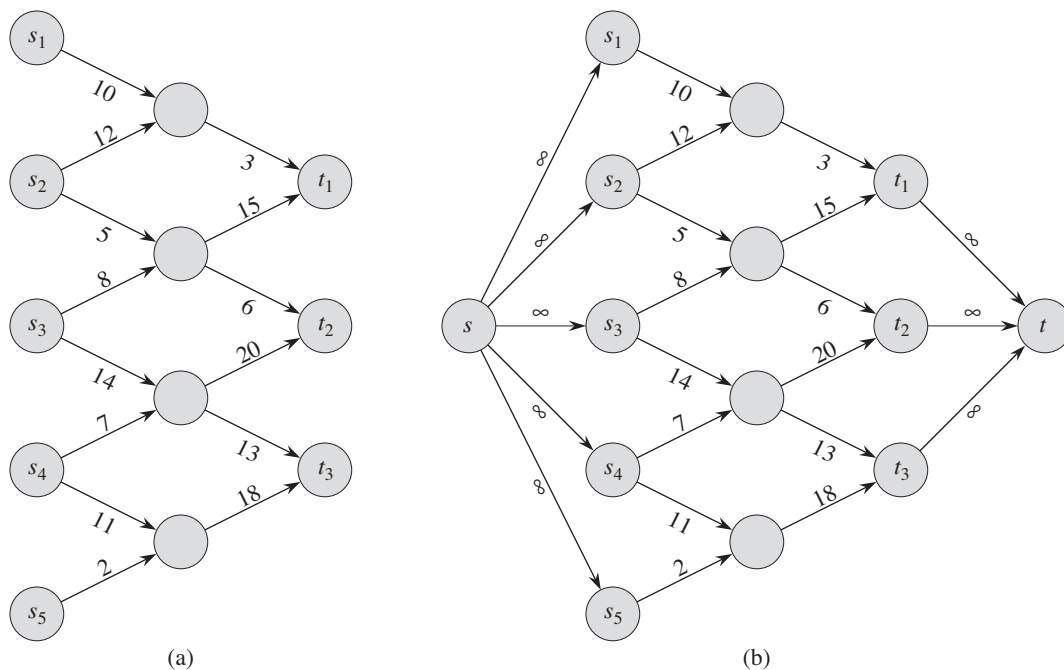


Figure 26.3 Converting a multiple-source, multiple-sink maximum-flow problem into a problem with a single source and a single sink. (a) A flow network with five sources $S = \{s_1, s_2, s_3, s_4, s_5\}$ and three sinks $T = \{t_1, t_2, t_3\}$. (b) An equivalent single-source, single-sink flow network. 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 .

26.1-2

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.

26.1-3

Suppose that a flow network $G = (V, E)$ violates the assumption that the network contains a path $s \rightsquigarrow v \rightsquigarrow t$ for all vertices $v \in V$. Let u be a vertex for which there is no path $s \rightsquigarrow u \rightsquigarrow t$. Show that there must exist a maximum flow f in G such that $f(u, v) = f(v, u) = 0$ for all vertices $v \in V$.

26.1-4

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 \mathbb{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 $0 \leq \alpha \leq 1$.

26.1-5

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

26.1-6

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 whether both his children can go to the same school as a maximum-flow problem.

26.1-7

Suppose that, in addition to edge capacities, a flow network has *vertex capacities*. That is each vertex v has a limit $l(v)$ on how much flow can pass through v . Show how to transform a flow network $G = (V, E)$ with vertex capacities into an equivalent flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G . How many vertices and edges does G' have?

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 26.6), 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 iteratively increases the value of the flow. 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 in G by finding an “augmenting path” in an associated “residual network” G_f . Once we know the edges of an augmenting path in G_f , we can easily identify specific edges in G for which we can change the flow so that we increase the value of the flow. Although each iteration of the Ford-Fulkerson method increases the value of the flow, we shall see that the flow on any particular edge of G may increase or decrease; decreasing the flow on some edges may be necessary in order to enable an algorithm to send more flow from the source to the sink. We repeatedly augment the flow until the residual network has no more augmenting paths. 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  $p$  in the residual network  $G_f$ 
3      augment flow  $f$  along  $p$ 
4  return  $f$ 
```

In order to implement and analyze the Ford-Fulkerson method, we need to introduce several additional concepts.

Residual networks

Intuitively, given a flow network G and a flow f , the residual network G_f consists of edges with capacities that represent how we can change the flow on edges of G . An edge of the flow network can admit an amount of additional flow equal to the edge’s capacity minus the flow on that edge. If that value is positive, we place that edge into G_f with a “residual capacity” of $c_f(u, v) = c(u, v) - f(u, v)$. The only edges of G that are in G_f are those that can admit more flow; those edges (u, v) whose flow equals their capacity have $c_f(u, v) = 0$, and they are not in G_f .

The residual network G_f may also contain edges that are not in G , however. As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge. In order to represent a possible decrease of a positive flow $f(u, v)$ on an edge in G , we place an edge (v, u) into G_f with residual capacity $c_f(v, u) = f(u, v)$ —that is, an edge that can admit flow in the opposite direction to (u, v) , at most canceling out the flow on (u, v) . These reverse edges in the residual network allow an algorithm to send back flow

it has already sent along an edge. Sending flow back along an edge is equivalent to *decreasing* the flow on the edge, which is a necessary operation in many algorithms.

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 $u, v \in V$. We define the **residual capacity** $c_f(u, v)$ by

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E, \\ f(v, u) & \text{if } (v, u) \in E, \\ 0 & \text{otherwise.} \end{cases} \quad (26.2)$$

Because of our assumption that $(u, v) \in E$ implies $(v, u) \notin E$, exactly one case in equation (26.2) applies to each ordered pair of vertices.

As an example of equation (26.2), if $c(u, v) = 16$ and $f(u, v) = 11$, then we can increase $f(u, v)$ by up to $c_f(u, v) = 5$ units before we exceed the capacity constraint on edge (u, v) . We also wish to allow an algorithm to return up to 11 units of flow from v to u , and hence $c_f(v, u) = 11$.

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

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}. \quad (26.3)$$

That is, as promised above, each edge of the residual network, or **residual edge**, can admit a flow that is greater than 0. Figure 26.4(a) repeats the flow network G and flow f of Figure 26.1(b), and Figure 26.4(b) shows the corresponding residual network G_f . The edges in E_f are either edges in E or their reversals, and thus

$$|E_f| \leq 2 |E|.$$

Observe that the residual network G_f is similar to a flow network with capacities given by c_f . It does not satisfy our definition of a flow network because it may contain both an edge (u, v) and its reversal (v, u) . Other than this difference, a residual network has the same properties as a flow network, and we can define a flow in the residual network as one that satisfies the definition of a flow, but with respect to capacities c_f in the network G_f .

A flow in a residual network provides a roadmap for adding flow to the original flow network. If f is a flow in G and f' is a flow in the corresponding residual network G_f , we define $f \uparrow f'$, the **augmentation** of flow f by f' , to be a function from $V \times V$ to \mathbb{R} , defined by

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{if } (u, v) \in E, \\ 0 & \text{otherwise.} \end{cases} \quad (26.4)$$

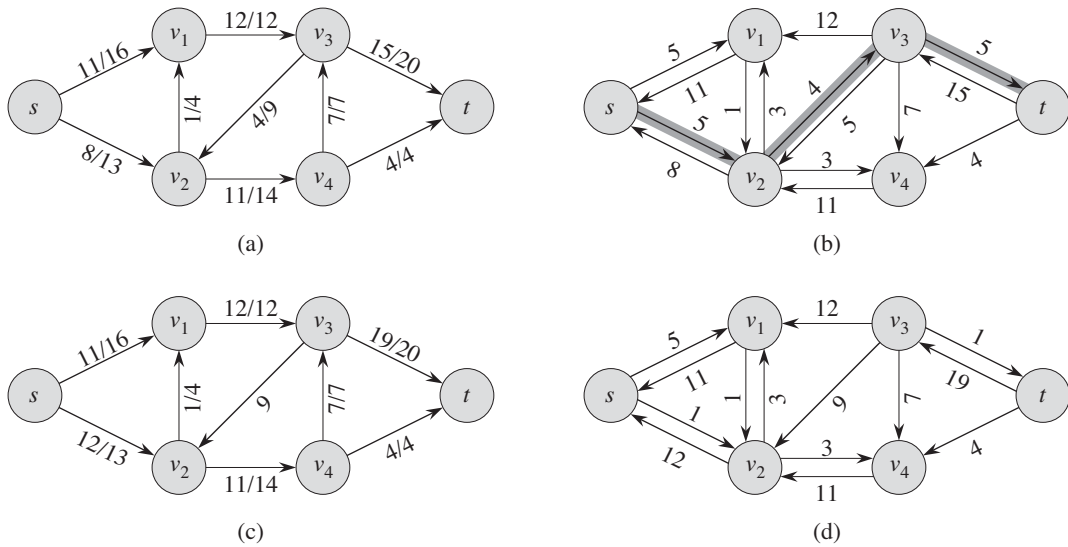


Figure 26.4 (a) The flow network G and flow f of Figure 26.1(b). (b) The residual network G_f with augmenting path p shaded; its residual capacity is $c_f(p) = c_f(v_2, v_3) = 4$. Edges with residual capacity equal to 0, such as (v_1, v_3) , are not shown, a convention we follow in the remainder of this section. (c) The flow in G that results from augmenting along path p by its residual capacity 4. Edges carrying no flow, such as (v_3, v_2) , are labeled only by their capacity, another convention we follow throughout. (d) The residual network induced by the flow in (c).

The intuition behind this definition follows the definition of the residual network. We increase the flow on (u, v) by $f'(u, v)$ but decrease it by $f'(v, u)$ because pushing flow on the reverse edge in the residual network signifies decreasing the flow in the original network. Pushing flow on the reverse edge in the residual network is also known as **cancellation**. For example, if we send 5 crates of hockey pucks from u to v and send 2 crates from v to u , we could equivalently (from the perspective of the final result) just send 3 crates from u to v and none from v to u . Cancellation of this type is crucial for any maximum-flow algorithm.

Lemma 26.1

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 function $f \uparrow f'$ defined in equation (26.4) is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Proof We first verify that $f \uparrow f'$ obeys the capacity constraint for each edge in E and flow conservation at each vertex in $V - \{s, t\}$.

For the capacity constraint, first observe that if $(u, v) \in E$, then $c_f(v, u) = f(u, v)$. Therefore, we have $f'(v, u) \leq c_f(v, u) = f(u, v)$, and hence

$$\begin{aligned} (f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) \quad (\text{by equation (26.4)}) \\ &\geq f(u, v) + f'(u, v) - f(u, v) \quad (\text{because } f'(v, u) \leq f(u, v)) \\ &= f'(u, v) \\ &\geq 0. \end{aligned}$$

In addition,

$$\begin{aligned} (f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) \quad (\text{by equation (26.4)}) \\ &\leq f(u, v) + f'(u, v) \quad (\text{because flows are nonnegative}) \\ &\leq f(u, v) + c_f(u, v) \quad (\text{capacity constraint}) \\ &= f(u, v) + c(u, v) - f(u, v) \quad (\text{definition of } c_f) \\ &= c(u, v). \end{aligned}$$

For flow conservation, because both f and f' obey flow conservation, we have that for all $u \in V - \{s, t\}$,

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

where the third line follows from the second by flow conservation.

Finally, we compute the value of $f \uparrow f'$. Recall that we disallow antiparallel edges in G (but not in G_f), and hence for each vertex $v \in V$, we know that there can be an edge (s, v) or (v, s) , but never both. We define $V_1 = \{v : (s, v) \in E\}$ to be the set of vertices with edges from s , and $V_2 = \{v : (v, s) \in E\}$ to be the set of vertices with edges to s . We have $V_1 \cup V_2 \subseteq V$ and, because we disallow antiparallel edges, $V_1 \cap V_2 = \emptyset$. We now compute

$$\begin{aligned} |f \uparrow f'| &= \sum_{v \in V} (f \uparrow f')(s, v) - \sum_{v \in V} (f \uparrow f')(v, s) \\ &= \sum_{v \in V_1} (f \uparrow f')(s, v) - \sum_{v \in V_2} (f \uparrow f')(v, s), \end{aligned} \tag{26.5}$$

where the second line follows because $(f \uparrow f')(w, x)$ is 0 if $(w, x) \notin E$. We now apply the definition of $f \uparrow f'$ to equation (26.5), and then reorder and group terms to obtain

$$\begin{aligned}
 |f \uparrow f'| &= \sum_{v \in V_1} (f(s, v) + f'(s, v) - f'(v, s)) - \sum_{v \in V_2} (f(v, s) + f'(v, s) - f'(s, v)) \\
 &= \sum_{v \in V_1} f(s, v) + \sum_{v \in V_1} f'(s, v) - \sum_{v \in V_1} f'(v, s) \\
 &\quad - \sum_{v \in V_2} f(v, s) - \sum_{v \in V_2} f'(v, s) + \sum_{v \in V_2} f'(s, v) \\
 &= \sum_{v \in V_1} f(s, v) - \sum_{v \in V_2} f(v, s) \\
 &\quad + \sum_{v \in V_1} f'(s, v) + \sum_{v \in V_2} f'(s, v) - \sum_{v \in V_1} f'(v, s) - \sum_{v \in V_2} f'(v, s) \\
 &= \sum_{v \in V_1} f(s, v) - \sum_{v \in V_2} f(v, s) + \sum_{v \in V_1 \cup V_2} f'(s, v) - \sum_{v \in V_1 \cup V_2} f'(v, s). \quad (26.6)
 \end{aligned}$$

In equation (26.6), we can extend all four summations to sum over V , since each additional term has value 0. (Exercise 26.2-1 asks you to prove this formally.) We thus have

$$\begin{aligned}
 |f \uparrow f'| &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{v \in V} f'(s, v) - \sum_{v \in V} f'(v, s) \quad (26.7) \\
 &= |f| + |f'|. \quad \blacksquare
 \end{aligned}$$

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, we may increase the flow on an edge (u, v) of an augmenting path by up to $c_f(u, v)$ without violating the capacity constraint on whichever of (u, v) and (v, u) is in the original flow network G .

The shaded path in Figure 26.4(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 we leave as Exercise 26.2-7, makes the above argument more precise.

Lemma 26.2

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 $f_p : V \times V \rightarrow \mathbb{R}$ by

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p, \\ 0 & \text{otherwise.} \end{cases} \quad (26.8)$$

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

The following corollary shows that if we augment f by f_p , we get another flow in G whose value is closer to the maximum. Figure 26.4(c) shows the result of augmenting the flow f from Figure 26.4(a) by the flow f_p in Figure 26.4(b), and Figure 26.4(d) shows the ensuing residual network.

Corollary 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 . Let f_p be defined as in equation (26.8), and suppose that we augment f by f_p . Then the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Proof Immediate from Lemmas 26.1 and 26.2. ■

Cuts of flow networks

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until it has found a maximum flow. How do we know that when the algorithm terminates, we have actually found a maximum flow? The **max-flow min-cut** theorem, which 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** $f(S, T)$ across the cut (S, T) is defined to be

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u). \quad (26.9)$$

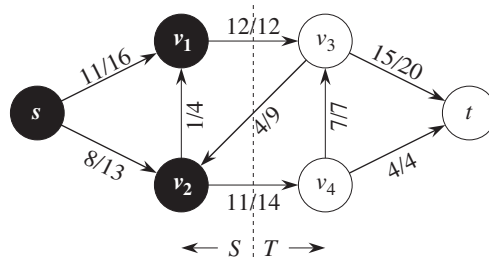


Figure 26.5 A cut (S, T) in the flow network of Figure 26.1(b), where $S = \{s, v_1, v_2\}$ and $T = \{v_3, v_4, t\}$. The vertices in S are black, and the vertices in T are white. The net flow across (S, T) is $f(S, T) = 19$, and the capacity is $c(S, T) = 26$.

The **capacity** of the cut (S, T) is

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v). \quad (26.10)$$

A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network.

The asymmetry between the definitions of flow and capacity of a cut is intentional and important. For capacity, we count only the capacities of edges going from S to T , ignoring edges in the reverse direction. For flow, we consider the flow going from S to T minus the flow going in the reverse direction from T to S . The reason for this difference will become clear later in this section.

Figure 26.5 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

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

and the capacity of this cut is

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

The following lemma shows that, for a given flow f , the net flow across any cut is the same, and it equals $|f|$, the value of the flow.

Lemma 26.4

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

Proof We can rewrite the flow-conservation condition for any node $u \in V - \{s, t\}$ as

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0. \quad (26.11)$$

Taking the definition of $|f|$ from equation (26.1) and adding the left-hand side of equation (26.11), which equals 0, summed over all vertices in $S - \{s\}$, gives

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right).$$

Expanding the right-hand summation and regrouping terms yields

$$\begin{aligned} |f| &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \sum_{v \in V} f(u, v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v, u) \\ &= \sum_{v \in V} \left(f(s, v) + \sum_{u \in S - \{s\}} f(u, v) \right) - \sum_{v \in V} \left(f(v, s) + \sum_{u \in S - \{s\}} f(v, u) \right) \\ &= \sum_{v \in V} \sum_{u \in S} f(u, v) - \sum_{v \in V} \sum_{u \in S} f(v, u). \end{aligned}$$

Because $V = S \cup T$ and $S \cap T = \emptyset$, we can split each summation over V into summations over S and T to obtain

$$\begin{aligned} |f| &= \sum_{v \in S} \sum_{u \in S} f(u, v) + \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &\quad + \left(\sum_{v \in S} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) \right). \end{aligned}$$

The two summations within the parentheses are actually the same, since for all vertices $x, y \in V$, the term $f(x, y)$ appears once in each summation. Hence, these summations cancel, and we have

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

A corollary to Lemma 26.4 shows how we can use cut capacities to bound the value of a flow.

Corollary 26.5

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 26.4 and the capacity constraint,

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

Corollary 26.5 yields the immediate consequence that the value of a maximum flow in a network is bounded from 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 26.6 (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 26.3, the flow found by augmenting f by f_p , where f_p is given by equation (26.8), 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 . Now consider a pair of vertices

$u \in S$ and $v \in T$. If $(u, v) \in E$, we must have $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$, which would place v in set S . If $(v, u) \in E$, we must have $f(v, u) = 0$, because otherwise $c_f(u, v) = f(v, u)$ would be positive and we would have $(u, v) \in E_f$, which would place v in S . Of course, if neither (u, v) nor (v, u) is in E , then $f(u, v) = f(v, u) = 0$. We thus have

$$\begin{aligned} f(S, T) &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &= \sum_{u \in S} \sum_{v \in T} c(u, v) - \sum_{v \in T} \sum_{u \in S} 0 \\ &= c(S, T). \end{aligned}$$

By Lemma 26.4, therefore, $|f| = f(S, T) = c(S, T)$.

(3) \Rightarrow (1): By Corollary 26.5, $|f| \leq c(S, T)$ 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 use p to modify the flow f . As Lemma 26.2 and Corollary 26.3 suggest, we replace f by $f \uparrow f_p$, obtaining a new flow whose value is $|f| + |f_p|$. The following implementation of the method computes the maximum flow in a flow network $G = (V, E)$ by updating the flow attribute $(u, v).f$ for each edge $(u, v) \in E$.¹ If $(u, v) \notin E$, we assume implicitly that $(u, v).f = 0$. We also assume that we are given the capacities $c(u, v)$ along with the flow network, and $c(u, v) = 0$ if $(u, v) \notin E$. We compute the residual capacity $c_f(u, v)$ in accordance with the formula (26.2). The expression $c_f(p)$ in the code is just a temporary variable that stores the residual capacity of the path p .

FORD-FULKERSON(G, s, t)

```

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

¹Recall from Section 22.1 that we represent an attribute f for edge (u, v) with the same style of notation— $(u, v).f$ —that we use for an attribute of any other object.

The FORD-FULKERSON algorithm simply expands on the FORD-FULKERSON-METHOD pseudocode given earlier. Figure 26.6 shows the result of each iteration in a sample run. Lines 1–2 initialize the flow f to 0. The **while** loop of lines 3–8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$. Each residual edge in path p is either an edge in the original network or the reversal of an edge in the original network. Lines 6–8 update the flow in each case appropriately, adding flow when the residual edge is an original edge and subtracting it otherwise. 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 we find the augmenting path p in line 3. If we choose it 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 we find the augmenting path by using a breadth-first search (which we saw in Section 22.2), however, the algorithm runs in polynomial time. Before proving this result, we obtain a simple bound for the case in which we choose the augmenting path arbitrarily and all capacities are integers.

In practice, the maximum-flow problem often arises with integral capacities. If the capacities are rational numbers, we can apply an appropriate scaling transformation to make them all integral. If f^* denotes a maximum flow in the transformed network, then a **straightforward** implementation of FORD-FULKERSON executes the **while loop of lines 3–8 at most $|f^*|$ times**, since the flow value increases by at least one unit in each iteration.

We can perform the work done within the **while** loop efficiently if we implement the flow network $G = (V, E)$ with the right data structure and find an augmenting path by a linear-time algorithm. 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 edges in G' , and therefore we can easily 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 $c_f(u, v) > 0$, where c_f conforms to equation (26.2). 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, as does the initialization in lines 1–2, making the total running time of the **FORD-FULKERSON algorithm $O(E |f^*|)$** .

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

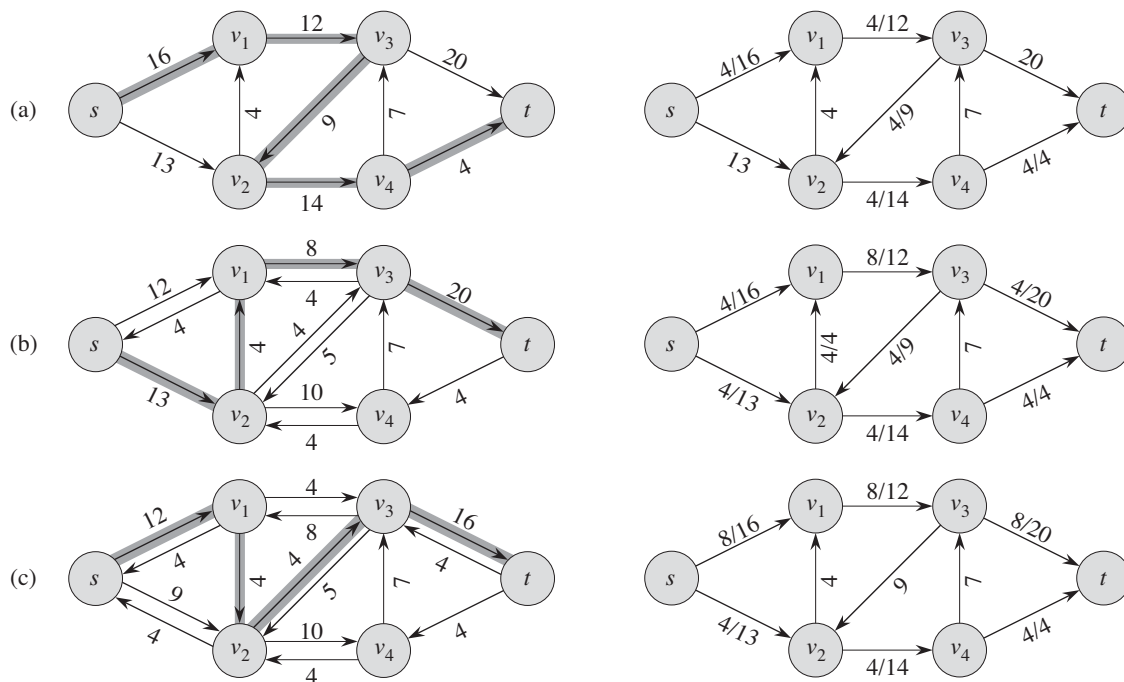


Figure 26.6 The execution of the basic Ford-Fulkerson algorithm. (a)–(e) Successive iterations of the **while** loop. The left side of each part shows the residual network G_f from line 3 with a shaded augmenting path p . The right side of each part shows the new flow f that results from augmenting f by f_p . The residual network in (a) is the input network G .

When the capacities are integral and the optimal flow value $|f^*|$ is small, the running time of the Ford-Fulkerson algorithm is good. Figure 26.7(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 $s \rightarrow u \rightarrow t$, and another 1,000,000 units traverse the path $s \rightarrow v \rightarrow t$. If the first augmenting path found by FORD-FULKERSON is $s \rightarrow u \rightarrow v \rightarrow t$, shown in Figure 26.7(a), the flow has value 1 after the first iteration. The resulting residual network appears in Figure 26.7(b). If the second iteration finds the augmenting path $s \rightarrow v \rightarrow u \rightarrow t$, as shown in Figure 26.7(b), the flow then has value 2. Figure 26.7(c) shows the resulting residual network. We can continue, choosing the augmenting path $s \rightarrow u \rightarrow v \rightarrow t$ in the odd-numbered iterations and the augmenting path $s \rightarrow v \rightarrow u \rightarrow t$ 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.

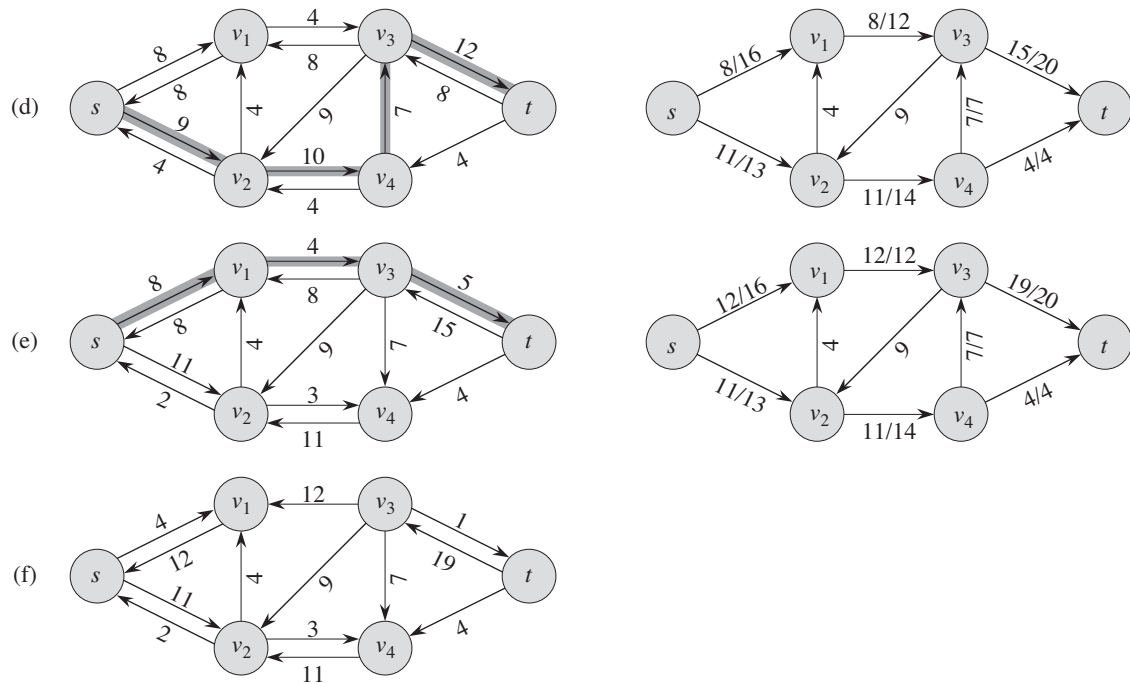


Figure 26.6, continued (f) The residual network at the last **while** loop test. It has no augmenting paths, and the flow f shown in (e) is therefore a maximum flow. The value of the maximum flow found is 23.

The Edmonds-Karp algorithm

We can improve the bound on FORD-FULKERSON by finding the augmenting path p in line 3 with a breadth-first search. That is, we choose the augmenting path as 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 26.7

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.

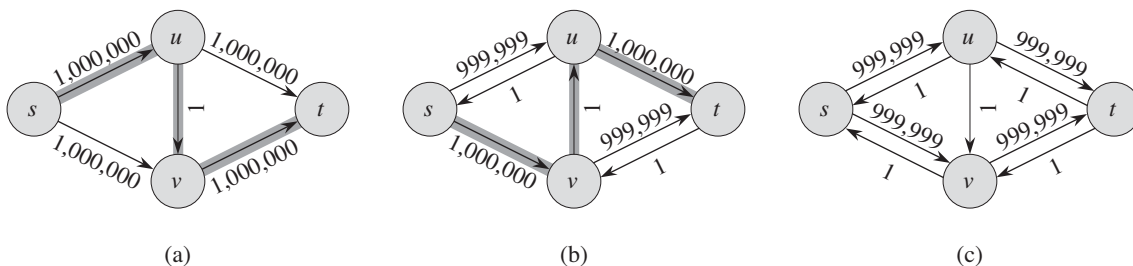


Figure 26.7 (a) A flow network for which FORD-FULKERSON can take $\Theta(E |f^*|)$ time, where f^* is a maximum flow, shown here with $|f^*| = 2,000,000$. The shaded path is an augmenting path with residual capacity 1. (b) The resulting residual network, with another augmenting path whose residual capacity is 1. (c) The resulting residual network.

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 $p = s \rightsquigarrow u \rightarrow v$ be a shortest path from s to v in $G_{f'}$, so that $(u, v) \in E_{f'}$ and

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1. \quad (26.12)$$

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

$$\delta_{f'}(s, u) \geq \delta_f(s, u). \quad (26.13)$$

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

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 && \text{(by Lemma 24.10, the triangle inequality)} \\ &\leq \delta_{f'}(s, u) + 1 && \text{(by inequality (26.13))} \\ &= \delta_{f'}(s, v) && \text{(by equation (26.12))} , \end{aligned}$$

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,

$$\begin{aligned} \delta_f(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 && \text{(by inequality (26.13))} \\ &= \delta_{f'}(s, v) - 2 && \text{(by equation (26.12))} , \end{aligned}$$

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 26.8

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 26.7, 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 an augmenting 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 network, 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. ■

Because we can implement each iteration of FORD-FULKERSON in $O(E)$ time when we find the augmenting path 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

Prove that the summations in equation (26.6) equal the summations in equation (26.7).

26.2-2

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?

26.2-3

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

26.2-4

In the example of Figure 26.6, what is the minimum cut corresponding to the maximum flow shown? Of the augmenting paths appearing in the example, which one cancels flow?

26.2-5

Recall that the construction in Section 26.1 that converts a flow network with multiple sources and sinks 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 network with multiple sources and sinks have finite capacity.

26.2-6

Suppose that each source s_i in a flow network with multiple sources and sinks produces exactly p_i units of flow, so that $\sum_{v \in V} f(s_i, v) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $\sum_{v \in V} f(v, t_j) = q_j$, 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 26.2.

26.2-8

Suppose that we redefine the residual network to disallow edges into s . Argue that the procedure FORD-FULKERSON still correctly computes a maximum flow.

26.2-9

Suppose that both f and f' are flows in a network G and we compute flow $f \uparrow f'$. Does the augmented flow satisfy the flow conservation property? Does it satisfy the capacity constraint?

26.2-10

Show how to find a maximum flow in a network $G = (V, E)$ by a sequence of at most $|E|$ augmenting paths. (*Hint*: Determine the paths *after* finding the maximum flow.)

26.2-11

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 to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

26.2-12

Suppose that you are given a flow network G , and G has edges entering the source s . Let f be a flow in G in which one of the edges (v, s) entering the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. Give an $O(E)$ -time algorithm to compute f' , given f , and assuming that all edge capacities are integers.

26.2-13

Suppose that you wish to find, among all minimum cuts in a flow network G with integral capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G .

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. Some other combinatorial problems 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. In order to solve this problem, we shall take advantage of an integrality property provided by the Ford-Fulkerson method. We shall also see how to use the Ford-Fulkerson method 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 the 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: graphs in which the vertex set can be partitioned into $V = L \cup R$, 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 26.8 illustrates the notion of a matching in a bipartite graph.

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(c). 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

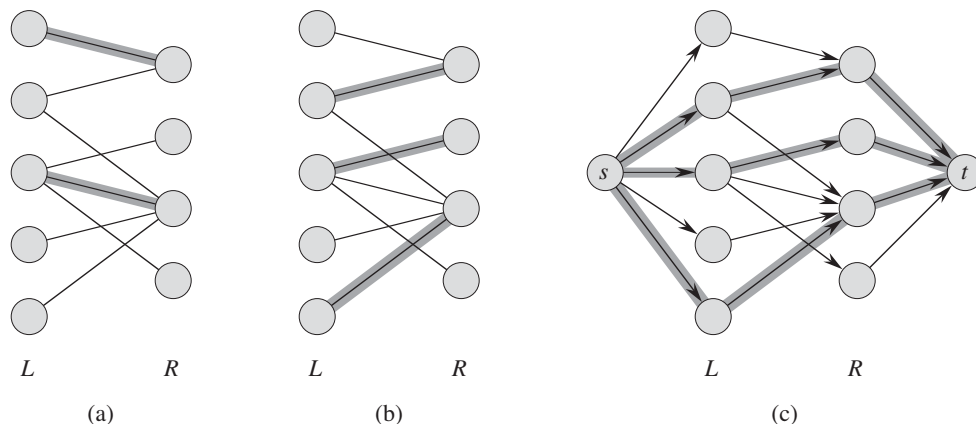


Figure 26.8 A bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$. (a) A matching with cardinality 2, indicated by shaded edges. (b) A maximum matching with cardinality 3. (c) The corresponding flow network G' with a maximum flow shown. Each edge has unit capacity. Shaded edges have a flow of 1, and all other edges carry no flow. The shaded edges from L to R correspond to those in the maximum matching from (b).

directed edges of G' are the edges of E , directed from L to R , along with $|V|$ new directed edges:

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in R\}.$$

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 26.9

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 $f(s, u) = f(u, v) = f(v, t) = 1$. For all other edges $(u, v) \in E'$, we define $f(u, v) = 0$. It is simple to verify that f satisfies the capacity constraint and flow conservation.

Intuitively, each edge $(u, v) \in M$ corresponds to one unit of flow in G' that traverses the path $s \rightarrow u \rightarrow v \rightarrow t$. 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 26.4, 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 flow entering it, and if one unit of flow does enter, by flow conservation, one unit of 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 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 applies to 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, $f(L \cup \{s\}, R \cup \{t\})$, the net flow across cut $(L \cup \{s\}, R \cup \{t\})$, is equal to $|M|$. Applying Lemma 26.4, we have that $|f| = f(L \cup \{s\}, R \cup \{t\}) = |M|$. ■

Based on Lemma 26.9, 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 26.10 (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 an integer. 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 26.9.

Corollary 26.11

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 26.9. 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 26.10, 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(c) 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 26.10.

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 ★

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 ★

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 $|L|$.

★ 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. Push-relabel methods also efficiently solve other flow problems, such as the minimum-cost flow problem. 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 $f : V \times V \rightarrow \mathbb{R}$ that satisfies the capacity constraint and the following relaxation of flow conservation:

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

for all vertices $u \in V - \{s\}$. That is, the flow into a vertex may exceed the flow out. We call the quantity

$$e(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \tag{26.14}$$

the **excess flow** into vertex u . The excess at a vertex is the amount by which the flow in exceeds the flow out. We say that a vertex $u \in V - \{s, t\}$ is **overflowing** if $e(u) > 0$.

We shall begin 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

You can understand the intuition behind the push-relabel method 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 out-flow pipe leading to an arbitrarily large reservoir that can accumulate fluid. Second, each vertex, its reservoir, and all its pipe connections sit on a platform whose height increases as the algorithm progresses.

Vertex heights determine how flow is pushed: we push flow only 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 push it only downhill. We fix the height of the source at $|V|$ and the height of the sink 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, we eventually push it downhill.

We may eventually find 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 . We increase its height 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, it has at least one outgoing pipe through which we can push more flow.

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 we have emptied all the reservoirs, the preflow is not only a “legal” flow, it is also a maximum flow.

The basic operations

From the preceding discussion, we see that a push-relabel algorithm performs **two** basic operations: **pushing flow excess from a vertex to one of its neighbors** and **relabeling a vertex**. The situations in which these operations apply depend 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 $h : V \rightarrow \mathbb{N}$ is a **height function**³ if $h(s) = |V|$, $h(t) = 0$, and

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

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

Lemma 26.12

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 network. ■

The push operation

The basic operation $\text{PUSH}(u, v)$ applies if **u is an overflowing vertex**, $c_f(u, v) > 0$, and $h(u) = h(v) + 1$. The pseudocode below updates the preflow f and the excess flows for u and v . It assumes that we can compute residual capacity $c_f(u, v)$ in constant time given c and f . We maintain the excess flow stored at a vertex u as the attribute $u.e$ and the height of u as the attribute $u.h$. The expression $\Delta_f(u, v)$ is a temporary variable that stores the amount of flow that we can push from u to v .

³In 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  $u.h = v.h + 1$ .
2  // Action: Push  $\Delta_f(u, v) = \min(u.e, c_f(u, v))$  units of flow from  $u$  to  $v$ .
3   $\Delta_f(u, v) = \min(u.e, c_f(u, v))$ 
4  if  $(u, v) \in E$ 
5       $(u, v).f = (u, v).f + \Delta_f(u, v)$ 
6  else  $(v, u).f = (v, u).f - \Delta_f(u, v)$ 
7   $u.e = u.e - \Delta_f(u, v)$ 
8   $v.e = v.e + \Delta_f(u, v)$ 

```

The code for PUSH operates as follows. Because vertex u has a positive excess $u.e$ and the residual capacity of (u, v) is positive, we can increase the flow from u to v by $\Delta_f(u, v) = \min(u.e, c_f(u, v))$ without causing $u.e$ to become negative or the capacity $c(u, v)$ to be exceeded. Line 3 computes the value $\Delta_f(u, v)$, and lines 4–6 update f . Line 5 increases the flow on edge (u, v) , because we are pushing flow over a residual edge that is also an original edge. Line 6 **decreases** the flow on edge (v, u) , because the **residual edge is actually the reverse** of an edge in the original network. Finally, lines 7–8 update the excess flows into vertices u and v . 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 $u.h = v.h + 1$. Thus, we push excess flow downhill only by a height differential of 1. By Lemma 26.12, 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, we would gain nothing 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) in the residual network becomes **saturated** ($c_f(u, v) = 0$ afterward); otherwise, it is a **nonsaturating push**. If an edge becomes saturated, it disappears from the residual network. A simple lemma characterizes one result of a nonsaturating push.

Lemma 26.13

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 $\Delta_f(u, v)$ actually pushed must equal $u.e$ prior to the push. Since $u.e$ is reduced by this amount, it becomes 0 after the push. ■

The *relabel* operation

The basic operation $\text{RELABEL}(u)$ applies if u is overflowing and if $u.h \leq v.h$ 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, and so s and t are ineligible for relabeling.)

$\text{RELABEL}(u)$

- 1 **// Applies when:** u is overflowing and for all $v \in V$ such that $(u, v) \in E_f$, we have $u.h \leq v.h$.
- 2 **// Action:** Increase the height of u .
- 3 $u.h = 1 + \min \{v.h : (u, v) \in E_f\}$

When we call the operation $\text{RELABEL}(u)$, we say that **vertex u is *relabelled***. 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, which in turn tells us that

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

Since all flows are nonnegative, we must therefore have at least one vertex v such that $(v, u).f > 0$. But then, $c_f(u, v) > 0$, which implies that $(u, v) \in E_f$. The operation $\text{RELABEL}(u)$ 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.

$\text{INITIALIZE-PREFLOW}(G, s)$

- 1 **for** each vertex $v \in G.V$
- 2 $v.h = 0$
- 3 $v.e = 0$
- 4 **for** each edge $(u, v) \in G.E$
- 5 $(u, v).f = 0$
- 6 $s.h = |G.V|$
- 7 **for** each vertex $v \in s.Adj$
- 8 $(s, v).f = c(s, v)$
- 9 $v.e = c(s, v)$
- 10 $s.e = s.e - c(s, v)$

INITIALIZE-PREFLOW creates an initial preflow f defined by

$$(u, v).f = \begin{cases} c(u, v) & \text{if } u = s, \\ 0 & \text{otherwise.} \end{cases} \quad (26.15)$$

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

$$u.h = \begin{cases} |V| & \text{if } u = s, \\ 0 & \text{otherwise.} \end{cases} \quad (26.16)$$

Equation (26.16) defines a height function because the only edges (u, v) for which $u.h > v.h + 1$ 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 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 26.14 (*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 $h(u) \leq h(v) + 1$ because h is a height function. If a push operation does not apply to an overflowing vertex u , then for all residual edges (u, v) , we must have $h(u) < h(v) + 1$, which implies $h(u) \leq h(v)$. Thus, a relabel operation applies 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 26.15 (Vertex heights never decrease)

During the execution of the GENERIC-PUSH-RELABEL procedure on a flow network $G = (V, E)$, for each vertex $u \in V$, the height $u.h$ never decreases. Moreover, whenever a relabel operation is applied to a vertex u , its height $u.h$ 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 $u.h \leq v.h$. Thus, $u.h < 1 + \min \{v.h : (u, v) \in E_f\}$, and so the operation must increase $u.h$. ■

Lemma 26.16

Let $G = (V, E)$ be a flow network with source s and sink t . Then the execution of GENERIC-PUSH-RELABEL on G maintains the attribute h 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 $u.h \leq v.h + 1$ afterward. Now consider a residual edge (w, u) that enters u . By Lemma 26.15, $w.h \leq u.h + 1$ before the operation RELABEL(u) implies $w.h < u.h + 1$ 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 $v.h = u.h - 1 < u.h + 1$, and so h remains a height function. In the latter case, removing (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 26.17

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 G_f contains a path p from s to t , where $p = \langle v_0, v_1, \dots, v_k \rangle$, $v_0 = s$, and $v_k = t$. Without loss of generality, p is a simple path, and so $k < |V|$. For $i = 0, 1, \dots, k - 1$, 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 $i = 0, 1, \dots, k - 1$. Combining these inequalities over path p yields $h(s) \leq h(t) + k$. But because $h(t) = 0$,

we have $h(s) \leq k < |V|$, 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 26.18 (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 739, 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 Lemma 26.14 and the invariant that f is always a preflow, there are no overflowing vertices. Therefore, f is a flow. Lemma 26.16 shows that h is a height function at termination, and thus Lemma 26.17 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 26.6), 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. We bound separately each of the three types of operations: relabels, saturating pushes, and nonsaturating pushes. 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. Recall that we allow edges into the source in the residual network.

Lemma 26.19

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 x , there is a simple path from x to s in the residual network G_f .

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

We take the definition of excess from equation (26.14), sum over all vertices in U , and note that $V = U \cup \overline{U}$, to obtain

$$\begin{aligned}
 \sum_{u \in U} e(u) &= \sum_{u \in U} \left(\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \right) \\
 &= \sum_{u \in U} \left(\left(\sum_{v \in U} f(v, u) + \sum_{v \in \overline{U}} f(v, u) \right) - \left(\sum_{v \in U} f(u, v) + \sum_{v \in \overline{U}} f(u, v) \right) \right) \\
 &= \sum_{u \in U} \sum_{v \in U} f(v, u) + \sum_{u \in U} \sum_{v \in \overline{U}} f(v, u) - \sum_{u \in U} \sum_{v \in U} f(u, v) - \sum_{u \in U} \sum_{v \in \overline{U}} f(u, v) \\
 &= \sum_{u \in U} \sum_{v \in \overline{U}} f(v, u) - \sum_{u \in U} \sum_{v \in \overline{U}} f(u, v) .
 \end{aligned}$$

We know that the quantity $\sum_{u \in U} e(u)$ must be positive because $e(x) > 0$, $x \in U$, all vertices other than s have nonnegative excess, and, by assumption, $s \notin U$. Thus, we have

$$\sum_{u \in U} \sum_{v \in \overline{U}} f(v, u) - \sum_{u \in U} \sum_{v \in \overline{U}} f(u, v) > 0 . \tag{26.17}$$

All edge flows are nonnegative, and so for equation (26.17) to hold, we must have $\sum_{u \in U} \sum_{v \in \overline{U}} f(v, u) > 0$. Hence, there must exist at least one pair of vertices $u' \in U$ and $v' \in \overline{U}$ with $f(v', u') > 0$. But, if $f(v', u') > 0$, there must be a residual edge (u', v') , which means that there is a simple path from x to v' (the path $x \rightsquigarrow u' \rightarrow v'$), thus contradicting the definition of U . ■

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

Lemma 26.20

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 $u.h \leq 2|V| - 1$ 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 $s.h = |V|$ and $t.h = 0$, both of which are no greater than $2|V| - 1$.

Now consider any vertex $u \in V - \{s, t\}$. Initially, $u.h = 0 \leq 2|V| - 1$. We shall show that after each relabeling operation, we still have $u.h \leq 2|V| - 1$. When u is

reabeled, it is overflowing, and Lemma 26.19 tells us that there is a simple path p from u to s in G_f . Let $p = \langle v_0, v_1, \dots, v_k \rangle$, where $v_0 = u$, $v_k = s$, and $k \leq |V| - 1$ because p is simple. For $i = 0, 1, \dots, k - 1$, we have $(v_i, v_{i+1}) \in E_f$, and therefore, by Lemma 26.16, $v_i.h \leq v_{i+1}.h + 1$. Expanding these inequalities over path p yields $u.h = v_0.h \leq v_k.h + k \leq s.h + (|V| - 1) = 2|V| - 1$. ■

Corollary 26.21 (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 $u.h$. The value of $u.h$ is initially 0 and by Lemma 26.20, it 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 26.20 also helps us to bound the number of saturating pushes.

Lemma 26.22 (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, $v.h = u.h - 1$. 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 $v.h = u.h + 1$. Since $u.h$ never decreases, in order for $v.h = u.h + 1$, the value of $v.h$ must increase by at least 2. Likewise, $u.h$ must increase by at least 2 between saturating pushes from v to u . Heights start at 0 and, by Lemma 26.20, 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 $u.h$ and $v.h$ 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 26.23 (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 $\Phi = \sum_{v: e(v) > 0} v.h$. 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 26.20, 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 26.13, 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 $u.h$, and it has increased by either 0 or $v.h$. Since $u.h - v.h = 1$, 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 Corollary 26.21 and Lemma 26.22 constrain the increase 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 26.24

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 26.21 and Lemmas 26.22 and 26.23. ■

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 26.25

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 26.4-2 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

Prove that, after the procedure INITIALIZE-PREFLOW(G, s) terminates, we have $s.e \leq -|f^*|$, where f^* is a maximum flow for G .

26.4-2

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 $O(V^2E)$.

26.4-3

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

26.4-4

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

26.4-5

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

26.4-6

Suppose that all edge capacities in a flow network $G = (V, E)$ are in the set $\{1, 2, \dots, k\}$. 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?)

26.4-7

Show that we could change line 6 of INITIALIZE-PREFLOW to

$$6 \quad s.h = |G.V| - 2$$

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

26.4-8

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

26.4-9 ★

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 to modify the generic push-relabel algorithm to maintain the property that $u.h < |V|$ implies $u.h = \delta_f(u, t)$ and that $u.h \geq |V|$ implies $u.h - |V| = \delta_f(u, s)$. The total time that your implementation dedicates to maintaining this property should be $O(VE)$.

26.4-10

Show that the number of nonsaturating pushes executed by the GENERIC-PUSH-RELABEL procedure 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 26.25. 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 even 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 we relabel a vertex, we move it 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 we can push flow. The following lemma shows that this network is a directed acyclic graph (dag).

Lemma 26.26 (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 $p = \langle v_0, v_1, \dots, v_k \rangle$, 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 $i = 1, 2, \dots, k$. 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 26.27

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 $\text{PUSH}(u, v)$ 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, we can push flow from u to v . Since u is overflowing, the operation $\text{PUSH}(u, v)$ applies. The only new residual edge that pushing flow from u to v can create is (v, u) . Since $v.h = u.h - 1$, 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 26.28

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 there are no admissible edges leaving u , then $\text{RELABEL}(u)$ 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 26.14, 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 $\text{RELABEL}(u)$ applies. After the relabel operation, $u.h = 1 + \min \{v.h : (u, v) \in E_f\}$. 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, $v.h = u.h + 1$ after the relabel, and so $v.h > u.h + 1$ just before the relabel. But by Lemma 26.12, 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** $u.N$ 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 $u.N$ if $(u, v) \in E$ or $(v, u) \in E$. The neighbor list $u.N$ contains exactly those vertices v for which there may be a residual edge (u, v) . The attribute $u.N.head$ points to the first vertex in $u.N$, and $v.next_neighbor$ points to the vertex following v in a neighbor list; 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 attribute $u.current$ points to the vertex currently under consideration in $u.N$. Initially, $u.current$ is set to $u.N.head$.

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

```

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 $u.N$.

1. If v is NIL, then we have run off the end of $u.N$. Line 4 relabels vertex u , and then line 5 resets the current neighbor of u to be the first one in $u.N$. (Lemma 26.29 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 .
3. If v is non-NIL but (u, v) is inadmissible, then line 8 advances $u.current$ one position further in the neighbor list $u.N$.

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 $u.e$ becomes zero, and neither the relabel operation nor advancing the pointer $u.current$ affects the value of $u.e$.

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

Lemma 26.29

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.

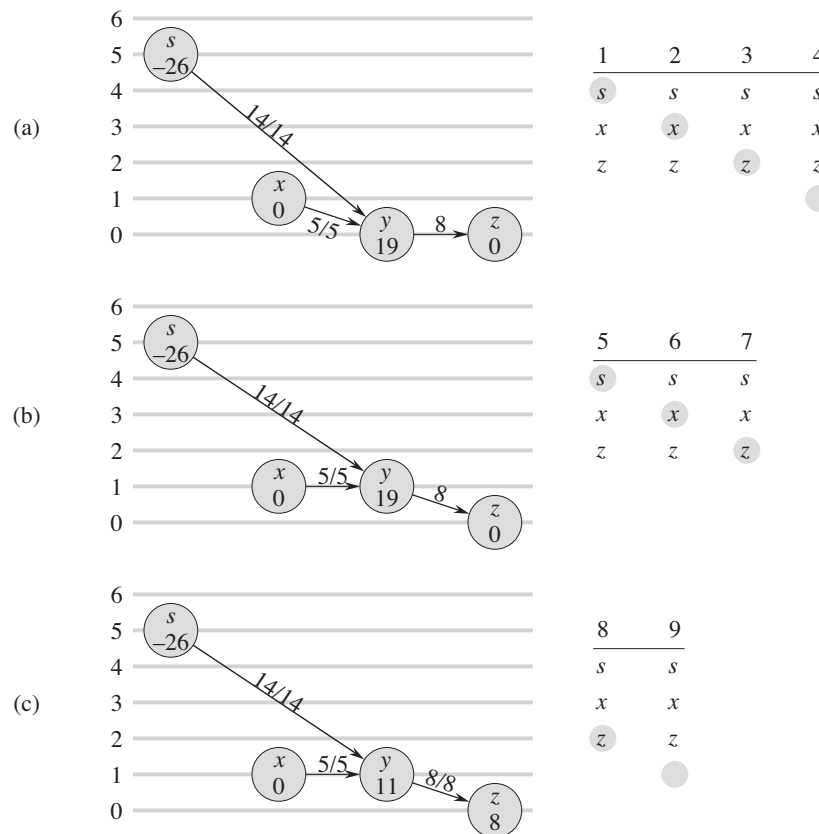


Figure 26.9 Discharging a vertex y . It takes 15 iterations of the **while** loop of DISCHARGE to push all the excess flow from y . Only the neighbors of y and edges of the flow network that enter or leave y are shown. In each part of the figure, the number inside each vertex is its excess at the beginning of the first iteration shown in the part, and each vertex is shown at its height throughout the part. The neighbor list $y.N$ at the beginning of each iteration appears on the right, with the iteration number on top. The shaded neighbor is $y.current$. **(a)** Initially, there are 19 units of excess to push from y , and $y.current = s$. Iterations 1, 2, and 3 just advance $y.current$, since there are no admissible edges leaving y . In iteration 4, $y.current = \text{NIL}$ (shown by the shading being below the neighbor list), and so y is relabeled and $y.current$ 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 iteration 7 pushes 8 units of excess flow from y to z . Because of the push, $y.current$ does not advance in this iteration. **(c)** Because the push in iteration 7 saturated edge (y, z) , it is found inadmissible in iteration 8. In iteration 9, $y.current = \text{NIL}$, and so vertex y is again relabeled and $y.current$ is reset.

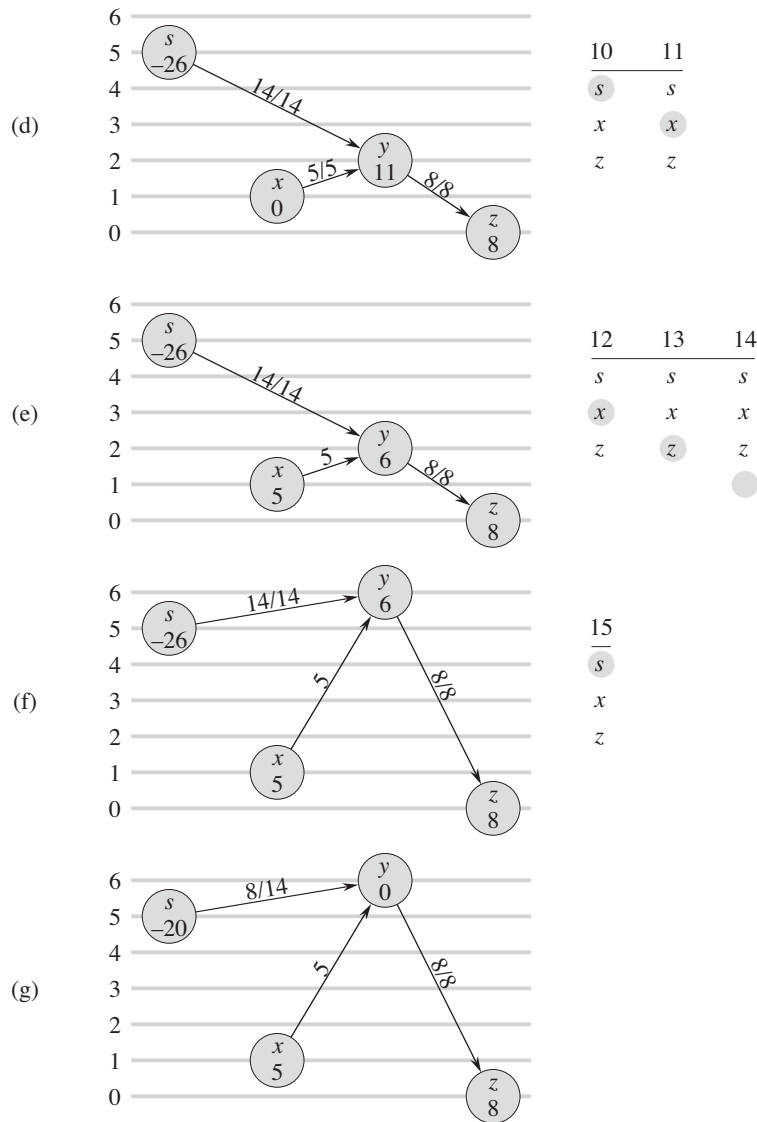


Figure 26.9, continued (d) In iteration 10, (y, s) is inadmissible, but iteration 11 pushes 5 units of excess flow from y to x . (e) Because $y.current$ did not advance in iteration 11, iteration 12 finds (y, x) to be inadmissible. Iteration 13 finds (y, z) inadmissible, and iteration 14 relabels vertex y and resets $y.current$. (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.

To prove the second statement, according to the test in line 1 and Lemma 26.28, we need only show that all edges leaving u are inadmissible. If a call to $\text{DISCHARGE}(u)$ starts with the pointer $u.\text{current}$ at the head of u 's neighbor list and finishes with it off the end of the list, then all of u 's outgoing edges are inadmissible and a relabel operation applies. It is possible, however, that during a call to $\text{DISCHARGE}(u)$, the pointer $u.\text{current}$ traverses only part of the list before the procedure returns. Calls to DISCHARGE on other vertices may then occur, but $u.\text{current}$ will continue moving through the list during the next call to $\text{DISCHARGE}(u)$. We now consider what happens during a complete pass through the list, which begins at the head of $u.N$ and finishes with $u.\text{current} = \text{NIL}$. Once $u.\text{current}$ reaches the end of the list, the procedure relabels u and begins a new pass. For the $u.\text{current}$ pointer to advance past a vertex $v \in u.N$ 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 26.27, pushes cannot create any admissible edges, regardless of which vertex the flow is pushed from. Thus, any admissible edge must be created by a relabel operation. But the vertex u is not relabeled during the pass, and by Lemma 26.28, any other vertex v that is relabeled during the pass (resulting from a call of $\text{DISCHARGE}(v)$) has no entering admissible edges after relabeling. Thus, at the end of the pass, all edges leaving u remain inadmissible, which completes the proof. ■

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 that follows. (Recall from Lemma 26.26 that the admissible network is a dag.)

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

```

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

```

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 Figure 26.10 illustrates, 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, line 8 discharges a vertex u . If u was relabeled by the DISCHARGE procedure, line 10 moves it to the front of list L . We can determine whether u was relabeled by comparing its height before the discharge operation, saved into the variable *old-height* in line 7, with its height afterward, in line 9. Line 11 makes the next iteration of the **while** loop use the vertex following u in list L . If line 10 moved u 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 operations only when they apply, since Lemma 26.29 guarantees that DISCHARGE performs them only 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, $s.h = |V|$ and $v.h = 0$ for all $v \in V - \{s\}$. Since $|V| \geq 2$ (because V contains at

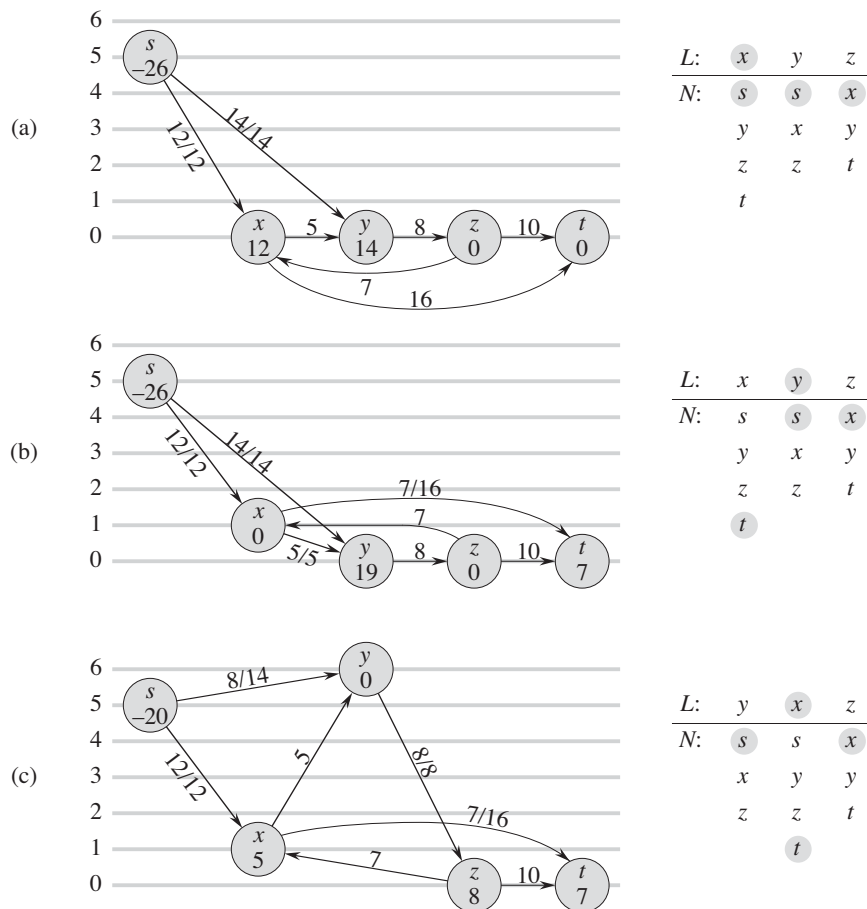


Figure 26.10 The action of RELABEL-TO-FRONT. (a) A flow network just before the first iteration of the **while** loop. Initially, 26 units of flow leave source s . On the right is shown the initial list $L = \langle x, y, z \rangle$, where initially $u = x$. Under each vertex in list L is its neighbor 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 moves 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 .

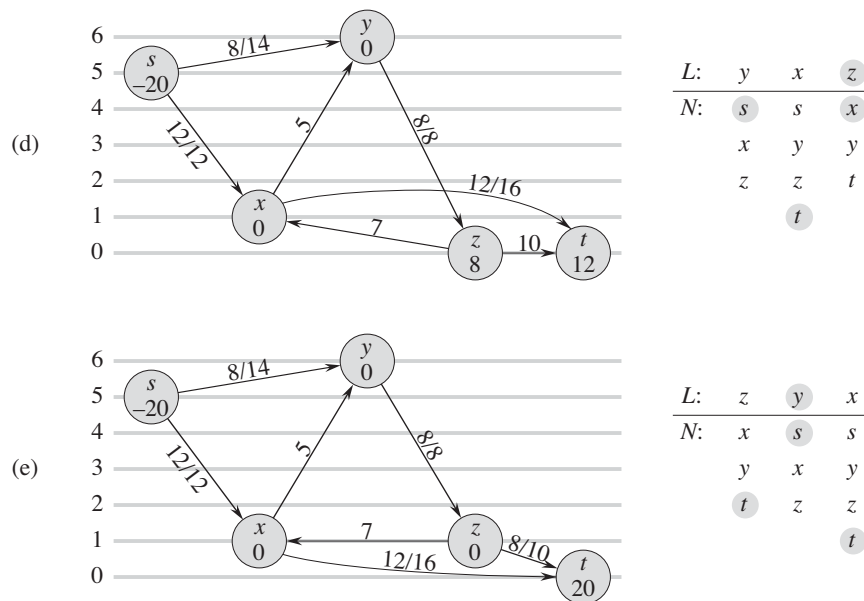


Figure 26.10, continued (d) Since vertex z follows vertex x in L , it is discharged. It is relabeled to height 1 and all 8 units of excess flow are pushed to t . Because z is relabeled, it moves to the front of L . (e) Vertex y now follows vertex z in L and is therefore discharged. But because y has 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.

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 $G_{f,h}$.

Because 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 each iteration of the **while** loop maintains the topological sort, we start by observing that the admissible network is changed only by push and relabel operations. By Lemma 26.27, push operations do not cause edges to become admissible. Thus, only relabel operations can create admissible edges. After a vertex u is relabeled, however, Lemma 26.28 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). When 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 just pointed out, 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 26.21, 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 26.4-3 provides an $O(VE)$ bound on the total time spent performing relabel operations, and Lemma 26.22 provides an $O(VE)$ bound on the total number of saturating push operations.

Theorem 26.30

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 we can see 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 26.4-3 provides an $O(VE)$ time bound on all the $O(V^2)$ relabels that are performed.

Now, suppose that the action updates the $u.current$ 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

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

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 how to implement this algorithm to compute a maximum flow in $O(V^3)$ time.

26.5-3

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

26.5-4 ★

Show that if we always discharge a highest overflowing vertex, we can make the push-relabel method 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 $v.h = k$. Show that all vertices with $v.h > k$ 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 $v.h > k$, to set $v.h = \max(v.h, |V| + 1)$. 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 i th row and the j th 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 $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ 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.

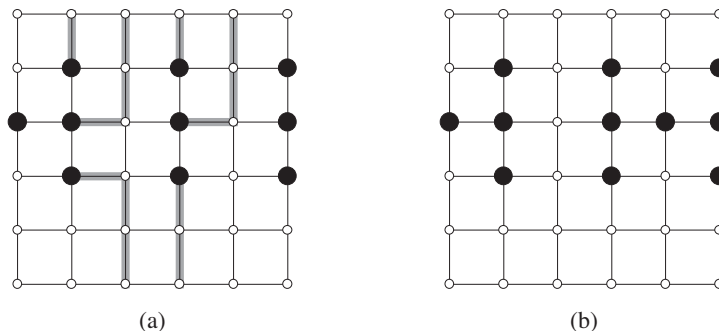


Figure 26.11 Grids for the escape problem. Starting points are black, and other grid vertices are white. **(a)** A grid with an escape, shown by shaded paths. **(b)** A grid with no escape.

- b.** 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 $V = \{1, 2, \dots, n\}$, construct the graph $G' = (V', E')$, where

$$V' = \{x_0, x_1, \dots, x_n\} \cup \{y_0, y_1, \dots, y_n\} ,$$

$$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\} ,$$

and run a maximum-flow algorithm.)

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

26-3 Algorithmic consulting

Professor Gore wants to open up an algorithmic consulting company. He has identified n important subareas of algorithms (roughly corresponding to different portions of this textbook), which he represents by the set $A = \{A_1, A_2, \dots, A_n\}$. In each subarea A_k , he can hire an expert in that area for c_k dollars. The consulting company has lined up a set $J = \{J_1, J_2, \dots, J_m\}$ of potential jobs. In order to perform job J_i , the company needs to have hired experts in a subset $R_i \subseteq A$ of

subareas. Each expert can work on multiple jobs simultaneously. If the company chooses to accept job J_i , it must have hired experts in all subareas in R_i , and it will take in revenue of p_i dollars.

Professor Gore's job is to determine which subareas to hire experts in and which jobs to accept in order to maximize the net revenue, which is the total income from jobs accepted minus the total cost of employing the experts.

Consider the following flow network G . It contains a source vertex s , vertices A_1, A_2, \dots, A_n , vertices J_1, J_2, \dots, J_m , and a sink vertex t . For $k = 1, 2, \dots, n$, the flow network contains an edge (s, A_k) with capacity $c(s, A_k) = c_k$, and for $i = 1, 2, \dots, m$, the flow network contains an edge (J_i, t) with capacity $c(J_i, t) = p_i$. For $k = 1, 2, \dots, n$ and $i = 1, 2, \dots, m$, if $A_k \in R_i$, then G contains an edge (A_k, J_i) with capacity $c(A_k, J_i) = \infty$.

- a. Show that if $J_i \in T$ for a finite-capacity cut (S, T) of G , then $A_k \in T$ for each $A_k \in R_i$.
- b. Show how to determine the maximum net revenue from the capacity of a minimum cut of G and the given p_i values.
- c. Give an efficient algorithm to determine which jobs to accept and which experts to hire. Analyze the running time of your algorithm in terms of m , n , and $r = \sum_{i=1}^m |R_i|$.

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 we increase the capacity of a single edge $(u, v) \in E$ by 1. Give an $O(V + E)$ -time algorithm to update the maximum flow.
- b. Suppose that we decrease the capacity of a single edge $(u, v) \in E$ 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 how to find an augmenting path of capacity at least K in $O(E)$ time, if such a path exists.

We can use the following modification of FORD-FULKERSON-METHOD to compute a maximum flow in G :

MAX-FLOW-BY-SCALING(G, s, t)

```

1   $C = \max_{(u,v) \in E} c(u, v)$ 
2  initialize flow  $f$  to 0
3   $K = 2^{\lceil \lg C \rceil}$ 
4  while  $K \geq 1$ 
5      while there exists an augmenting path  $p$  of capacity at least  $K$ 
6          augment flow  $f$  along  $p$ 
7       $K = K/2$ 
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 network G_f is at most $2K |E|$ each time line 4 is executed.
- e. Argue that the inner **while** loop of lines 5–6 executes $O(E)$ 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 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, \dots, P_k are vertex-disjoint augmenting paths with respect to M , then the symmetric difference $M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ is a matching with cardinality $|M| + k$.

The general structure of our algorithm is the following:

HOPCROFT-KARP(G)

```

1   $M = \emptyset$ 
2  repeat
3      let  $\mathcal{P} = \{P_1, P_2, \dots, P_k\}$  be a maximal set of vertex-disjoint
        shortest augmenting paths with respect to  $M$ 
4       $M = M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ 
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 $|M| \leq |M^*|$, 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, \dots, P_k be a maximal set of vertex-disjoint augmenting paths of length l with respect to M . Let $M' = M \oplus (P_1 \cup \dots \cup P_k)$, 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, \dots, P_k , then P has more than l edges.
- d. Now suppose that P is not vertex-disjoint from P_1, P_2, \dots, P_k . Let A be the set of edges $(M \oplus M') \oplus P$. Show that $A = (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P$ and that $|A| \geq (k + 1)l$. 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, \dots, 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 [103], Lawler [224], Papadimitriou and Steiglitz [271], and Tarjan [330] are good references for network flow and related algorithms. Goldberg, Tardos, and Tarjan [139] also provide a nice survey of algorithms for network-flow problems, and Schrijver [304] has written an interesting review of historical developments in the field of network flows.

The Ford-Fulkerson method is due to Ford and Fulkerson [109], 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 [102], and independently Dinic [89], proved that this strategy yields a polynomial-time algorithm. A related idea, that of using “blocking flows,” was also first developed by Dinic [89]. Karzanov [202] first developed the idea of preflows. The push-relabel method is due to Goldberg [136] and Goldberg and Tarjan [140]. 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 [62] proposed pushing flow from the overflowing vertex of maximum height. Cheriyan and Hagerup [61] suggested randomly permuting the neighbor lists, and several researchers [14, 204, 276] developed clever derandomizations of this idea, leading to a sequence of faster algorithms. The algorithm of King, Rao, and Tarjan [204] 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, by Goldberg and Rao [138], 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