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 $u.h \le \delta_f(u, t)$ and that $u.h \ge |V|$ implies $u.h - |V| \le \delta_f(u, s)$.

26.4-9 $\star$
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 $u.h \ge |V|$ 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 $|V| \ge 4$.

$\star$ 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 $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$ and $k > 0$. Since each edge in p is admissible, we have $h(v_{i-1}) = h(v_i) + 1$ for $i = 1, 2, \ldots, k$. Summing around the cycle gives

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

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

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

Lemma 26.27
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 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 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 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 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 $v.next\text{-}neighbor$ 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 == 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.

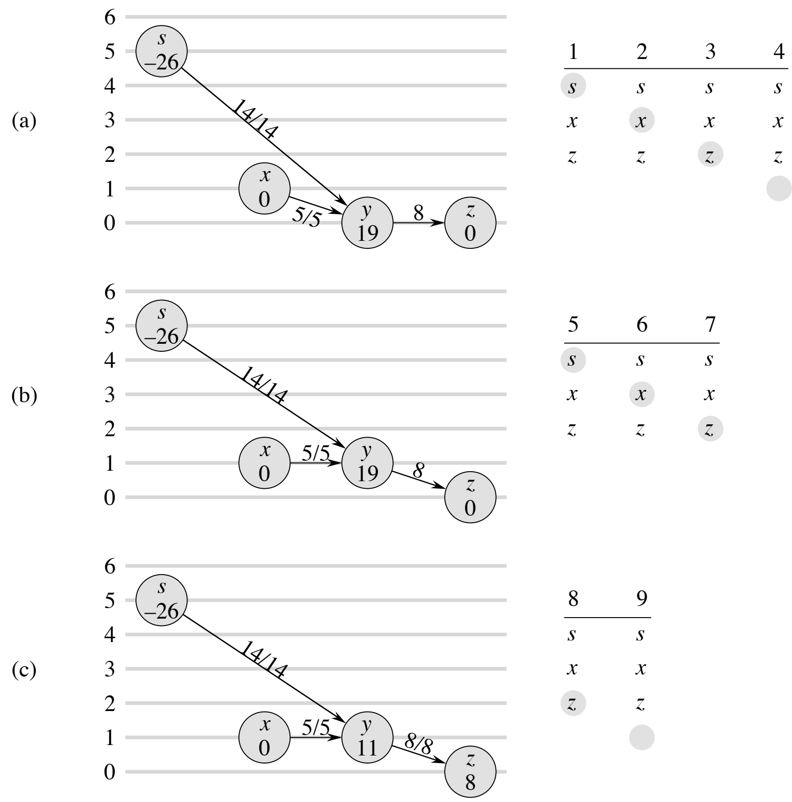

Figure 26.9 Discharging a vertex y. It takes 15 iterations of the **while** loop of DISCHARGE to push all the excess flow from y. Only the neighbors of y and edges of the flow network that enter or leave y are shown. In each part of the figure, the number inside each vertex is its excess at the beginning 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.

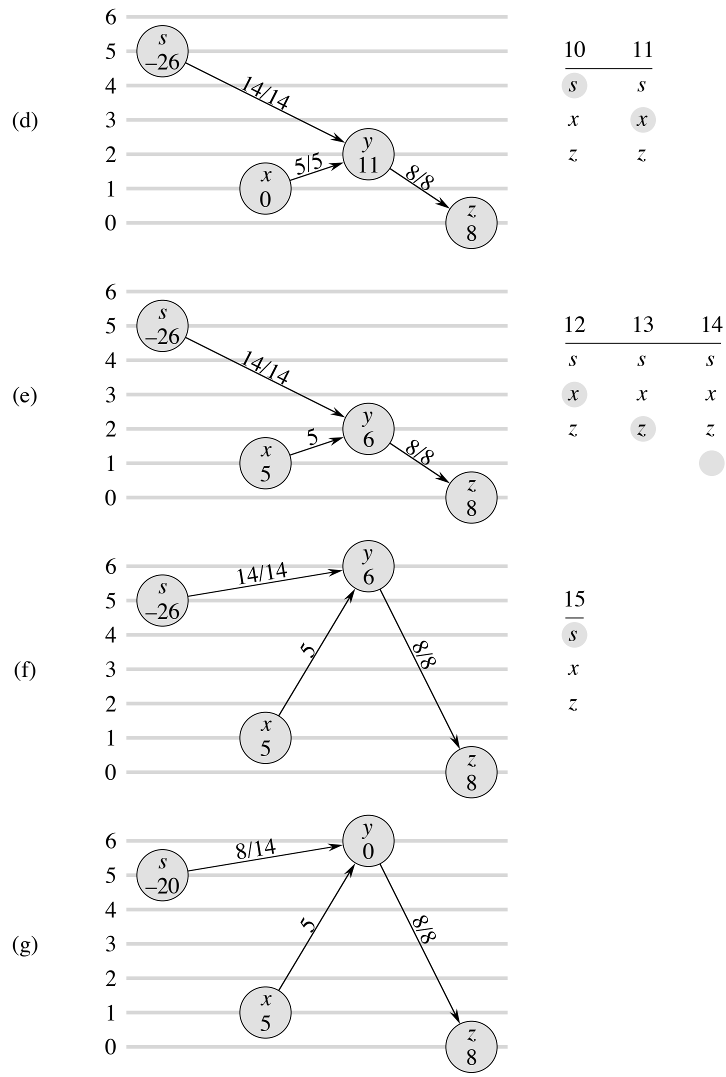

Figure 26.9, continued **(d)** In iteration 10, (y, s) is inadmissible, but iteration 11 pushes 5 units of excess flow from y to x. **(e)** Because $y.current$ did not advance in iteration 11, iteration 12 finds (y, x) to be inadmissible. Iteration 13 finds (y, z) inadmissible, and iteration 14 relabels 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 DISCHARGE(u) starts with the pointer $u.current$ 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 DISCHARGE(u), the pointer $u.current$ traverses only part of the list before the procedure returns. Calls to DISCHARGE on other vertices may then occur, but $u.current$ will continue moving through the list during the next call to 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 $u.current = \text{NIL}$. Once $u.current$ reaches the end of the list, the procedure relabels u and begins a new pass. For the $u.current$ 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 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 $u.next$ points to the vertex that follows u in list L and that, as usual, $u.next = \text{NIL}$ 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 ∈ G.V − {s, t}
 4      u.current = u.N.head
 5  u = L.head
 6  while u ≠ 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

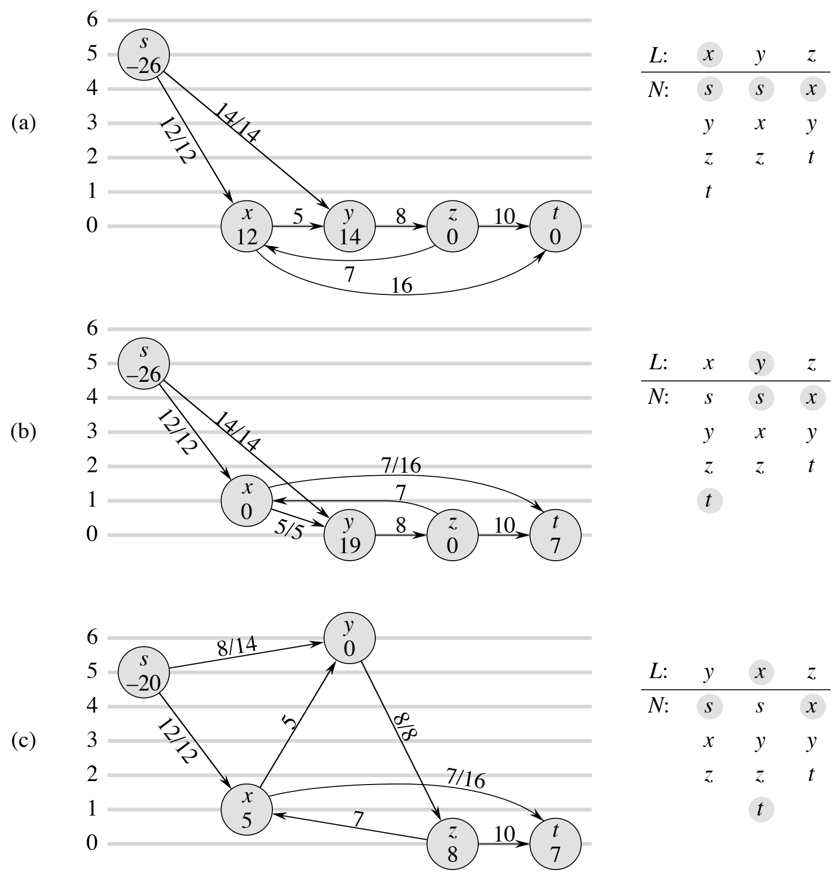

Figure 26.10 The action of RELABEL-TO-FRONT. **(a)** A flow network just before the first iteration of the **while** loop. Initially, 26 units of flow leave source s. On the right is shown the initial list $L = \langle x, y, z \rangle$, where initially $u = x$. Under each vertex in list L is its neighbor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

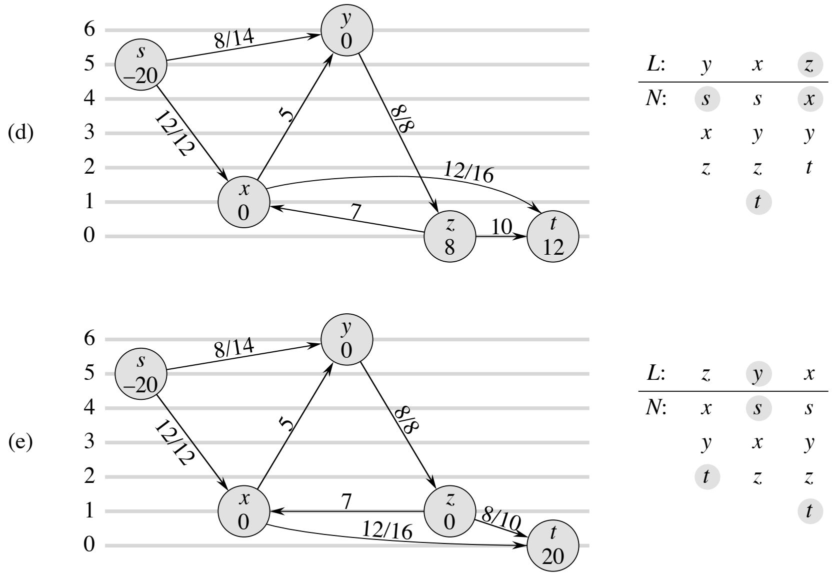

Figure 26.10, continued **(d)** Since vertex z follows vertex x in L, it is discharged. It is relabeled to height 1 and all 8 units of excess flow are pushed to t. Because z is relabeled, it moves to the front of L. **(e)** Vertex y now follows vertex z in L and is therefore discharged. But because y 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 $\langle \nu_1, \nu_2, \nu_3, \nu_4 \rangle$ and that the neighbor lists are

$$
\begin{aligned}
\nu_1.N &= \langle s, \nu_2, \nu_3 \rangle , \\
\nu_2.N &= \langle s, \nu_1, \nu_3, \nu_4 \rangle , \\
\nu_3.N &= \langle \nu_1, \nu_2, \nu_4, t \rangle , \\
\nu_4.N &= \langle \nu_2, \nu_3, t \rangle .
\end{aligned}
$$

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 $0 < k \le |V| - 1$ for which no vertex has $\nu.h = k$. Show that all vertices with $\nu.h > k$ are on the source side of a minimum cut. If such a k exists, the ***gap heuristic*** updates every vertex $\nu \in V - \{s\}$ for which $\nu.h > k$, to set $\nu.h = \max(\nu.h, |V| + 1)$. 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 ith row and the jth column by (i, j). All vertices in a grid have exactly four neighbors, except for the boundary vertices, which are the points (i, j) for which $i = 1, i = n, j = 1$, or $j = n$.

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

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