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 $O(V^{٢}E)$ bound given by Corollary ٢٦.٢٦. We shall now examine the relabel-to-front algorithm, a push-relabel algorithm whose running time is $O(V^{٣})$, which is asymptotically at least as good as $O(V^{٢}E)$, and better for dense networks.

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

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

Admissible edges and networks

If $G = (V, E)$ is a flow network with source *s* and sink *t*, *f* is a preflow in *G*, and *h* is a height function, then we say that (u, v) is an ***admissible edge*** if $c_f(u, v) > ٠$ and $h(u) = h(v) + ١$. Otherwise, (u, v) is ***inadmissible***. The ***admissible network*** is $G_{f,h} = (V, E_{f,h})$, where $E_{f,h}$ is the set of admissible edges.

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

Lemma ٢٦.٢٧: (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 $p = \langle v_٠, v_١, \ldots, v_k \rangle$, where $v_٠ = v_k$ and $k > ٠$. Since each edge in *p* is admissible, we have $h(v_{i-١}) = h(v_i) + ١$ for $i = ١, ٢, \ldots, k$. 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 $٠ = k$.

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

Lemma ٢٦.٢٨

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

Proof By the definition of an admissible edge, flow can be pushed from *u* to *v*. Since *u* is overflowing, the operation PUSH(u, v) applies. The only new residual edge that can be created by pushing flow from u to v is the edge (v, u). Since $h[v] = h[u] - ١$, edge (v, u) cannot become admissible. If the operation is asaturating push, then $c_f(u, v) = ٠$ afterward and (u, v) becomes inadmissible.

Lemma ٢٦.٢٩

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 ٢٦.١٥, either a push or a relabel operation applies to it. If there are no admissible edges leaving u, then no flow can be pushed from u and so RELABEL(u) applies. After the relabel operation, $h[u] = ١ + \min \{h[v] : (u, v) \in E_f\}$. Thus, if v is a vertex that realizes the minimum in this set, the edge (u, v) becomes admissible. Hence, after the relabel, there is at least one admissible edge leaving u.

To show that no admissible edges enter u after a relabel operation, suppose that there is a vertex v such that (v, u) is admissible. Then, $h[v] = h[u] + ١$ after the relabel, and so $h[v] > h[u] + ١$ just before the relabel. But by Lemma ٢٦.١٣, no residual edges exist between vertices whose heights differ by more than ١. Moreover, relabeling a vertex does not change the residual network. Thus, (v, u) is not in the residual network, and hence it cannot be in the admissible network.

Neighbor lists

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

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

Discharging an overflowing vertex

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

```
DISCHARGE(u)
١  while $e[u] > ٠$
٢     do $v \leftarrow current[u]$
٣        if $v$ = NIL
٤           then RELABEL($u$)
٥                $current[u] \leftarrow head[N[u]]$
٦        elseif $c_f(u, v) > ٠$ and $h[u] = h[v] + ١$
٧           then PUSH($u$, $v$)
```

```
٨        else current[u] ← next-neighbor[v]
```

Figure ٢٦.٩ steps through several iterations of the **while** loop of lines ١–٨, which executes as long as vertex *u* has positive excess. Each iteration performs exactly one of three actions, depending on the current vertex *v* in the neighbor list *N*[*u*].

١. If *v* is NIL, then we have run off the end of *N*[*u*]. Line ٤ relabels vertex *u*, and then line ٥ resets the current neighbor of *u* to be the first one in *N*[*u*]. (Lemma ٢٦.٣٠ below states that the relabel operation applies in this situation.)

٢. If *v* is non-NIL and (*u*, *v*) is an admissible edge (determined by the test in line ٦), then line ٧ pushes some (or possibly all) of *u*'s excess to vertex *v*.

٣. If *v* is non-NIL but (*u*, *v*) is inadmissible, then line ٨ advances *current*[*u*] one position further in the neighbor list *N*[*u*].

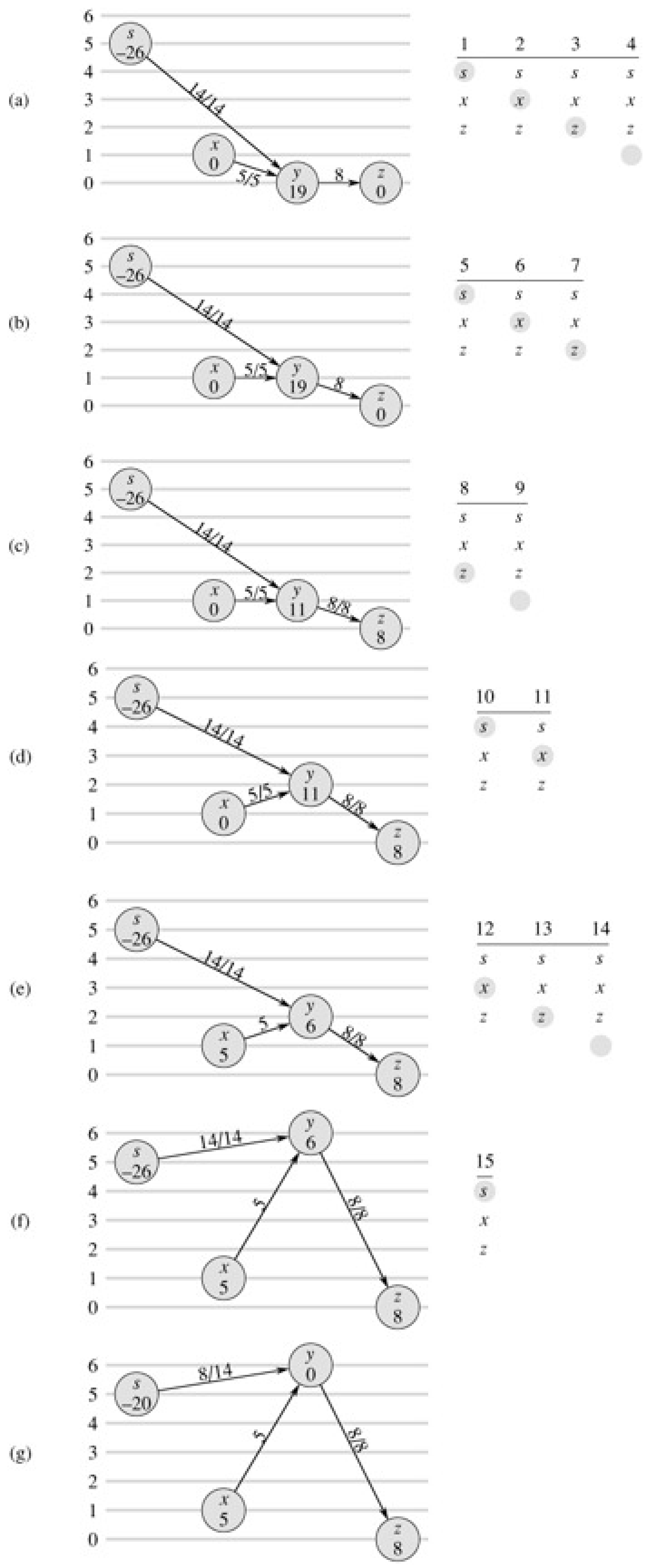
(a)
s −26
14/14
x 0
5/5
y 19
8
z 0
1 2 3 4
s s s s
x x x x
z z z z
(b)
s −26
14/14
x 0
5/5
y 19
8
z 0
5 6 7
s s s
x x x
z z z
(c)
s −26
14/14
x 0
5/5
y 11
8/8
z 8
8 9
s s
x x
z z
(d)
s −26
14/14
x 0
5/5
y 11
8/8
z 8
10 11
s s
x x
z z
(e)
s −26
14/14
x 5
5
y 6
8/8
z 8
12 13 14
s s s
x x x
z z z
(f)
s −26
14/14
y 6
5
x 5
8/8
z 8
15
s
x
z
(g)
s −20
8/14
y 0
5
x 5
8/8
z 8
0 1 2 3 4 5 6

Figure ٢٦.٩: Discharging a vertex *y*. It takes ١٥ iterations of the *while* loop of DISCHARGE to push all the excess flow from *y*. Only the neighbors of *y* and edges entering or leaving *y* are shown. In each part, the number inside each vertex is its excess at the beginning of the first iteration shown in the part, and each vertex is shown at its height throughout the part. To the right is shown the neighbor list *N*[*y*] at the beginning of each iteration, with the iteration number on top. The shaded neighbor is *current*[*y*]. *(a)* Initially, there are ١٩ units of excess to push from *y*, and *current*[*y*] = *s*. Iterations ١, ٢, and ٣ just advance *current*[*y*], since there are no admissible edges leaving *y*. In iteration ٤, *current*[*y*] = NIL (shown by the shading being below the neighbor list), and so *y* is relabeled and *current*[*y*] is reset to the head of the neighbor list. *(b)* After relabeling, vertex *y* has height ١. In iterations ٥ and ٦, edges (*y, s*) and (*y, x*) are found to be inadmissible, but ٨ units of excess flow are pushed from *y* to *z* in iteration ٧. Because of the push, *current*[*y*] is not advanced in this iteration. *(c)* Because the push in iteration ٧ saturated edge (*y, z*), it is found inadmissible in iteration ٨. In iteration ٩, *current*[*y*] = NIL, and so vertex *y* is again relabeled and *current*[*y*] is reset. *(d)* In iteration ١٠, (*y, s*) is inadmissible, but ٥ units of excess flow are pushed from *y* to *x* in iteration ١١. *(e)* Because *current*[*y*] was not advanced in iteration ١١, iteration ١٢ finds (*y, x*) to be inadmissible. Iteration ١٣ finds (*y, z*) inadmissible, and iteration ١٤ relabels vertex *y* and resets *current*[*y*]. *(f)* Iteration ١٥ pushes ٦ 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.

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

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

Lemma ٢٦.٣٠

If DISCHARGE calls PUSH(*u*, *v*) in line ٧, then a push operation applies to (*u*, *v*). If DISCHARGE calls RELABEL(*u*) in line ٤, then a relabel operation applies to *u*.

Proof The tests in lines ١ and ٦ ensure that a push operation occurs only if the operation applies, which proves the first statement in the lemma.

To prove the second statement, according to the test in line ١ and Lemma ٢٦.٢٩, we need only show that all edges leaving *u* are inadmissible. Observe that as DISCHARGE(*u*) is repeatedly called, the pointer *current*[*u*] moves down the list *N*[*u*]. Each "pass" begins at the head of *N*[*u*] and finishes with *current*[*u*] = NIL, at which point *u* is relabeled and a new pass begins. For the *current*[*u*] pointer to advance past a vertex $v \in N[u]$ during a pass, the edge (*u*, *v*) must be deemed inadmissible by the test in line ٦. 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 ٢٦.٢٨, pushes cannot create any admissible edges, let alone one leaving *u*. Thus, any admissible edge must be created by a relabel operation. But the vertex *u* is not relabeled during the pass, and by Lemma ٢٦.٢٩, any other vertex *v* that is relabeled during the pass has no entering admissible edges after relabeling. Thus, at the end of the pass, all edges leaving *u* remain inadmissible, and the lemma is proved.

The relabel-to-front algorithm

In the relabel-to-front algorithm, we maintain a linked list *L* consisting of all vertices in *V* - {*s*, *t*}. A key property is that the vertices in *L* are topologically sorted according to the admissible network, as we shall see in the loop invariant below. (Recall from Lemma ٢٦.٢٧ that the admissible network is a dag.)

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

```
RELABEL-TO-FRONT($G$, $s$, $t$)
١  INITIALIZE-PREFLOW($G$, $s$)
٢  $L \leftarrow V[G] - \{s, t\}$, in any order
٣  for each vertex $u \in V[G] - \{s, t\}$
٤      do $current[u] \leftarrow head[N[u]]$
٥  $u \leftarrow head[L]$
٦  while $u \neq$ NIL
٧     do $old\text{-}height \leftarrow h[u]$
٨        DISCHARGE($u$)
٩        if $h[u] > old\text{-}height$
١٠          then move $u$ to the front of list $L$
١١        $u \leftarrow next[u]$
```

The relabel-to-front algorithm works as follows. Line ١ initializes the preflow and heights to the same values as in the generic push-relabel algorithm. Line ٢ initializes the list *L* to contain all potentially overflowing vertices, in any order. Lines ٣–٤ initialize the *current* pointer of each vertex *u* to the first vertex in *u*'s neighbor list.

As shown in Figure ٢٦.١٠, the **while** loop of lines ٦–١١ runs through the list *L*, discharging vertices. Line ٥ makes it start with the first vertex in the list. Each time through the loop, a vertex *u* is discharged in line ٨. If *u* was relabeled by the DISCHARGE procedure, line ١٠ moves it to the front of list *L*. This determination is made by saving *u*'s height in the variable *old-height* before the discharge operation (line ٧) and comparing this saved height to *u*'s height afterward (line ٩). Line ١١ makes the next iteration of the **while** loop use the vertex following *u* in list *L*. If *u* was moved to the front of the list, the vertex used in the next iteration is the one following *u* in its new position in the list.

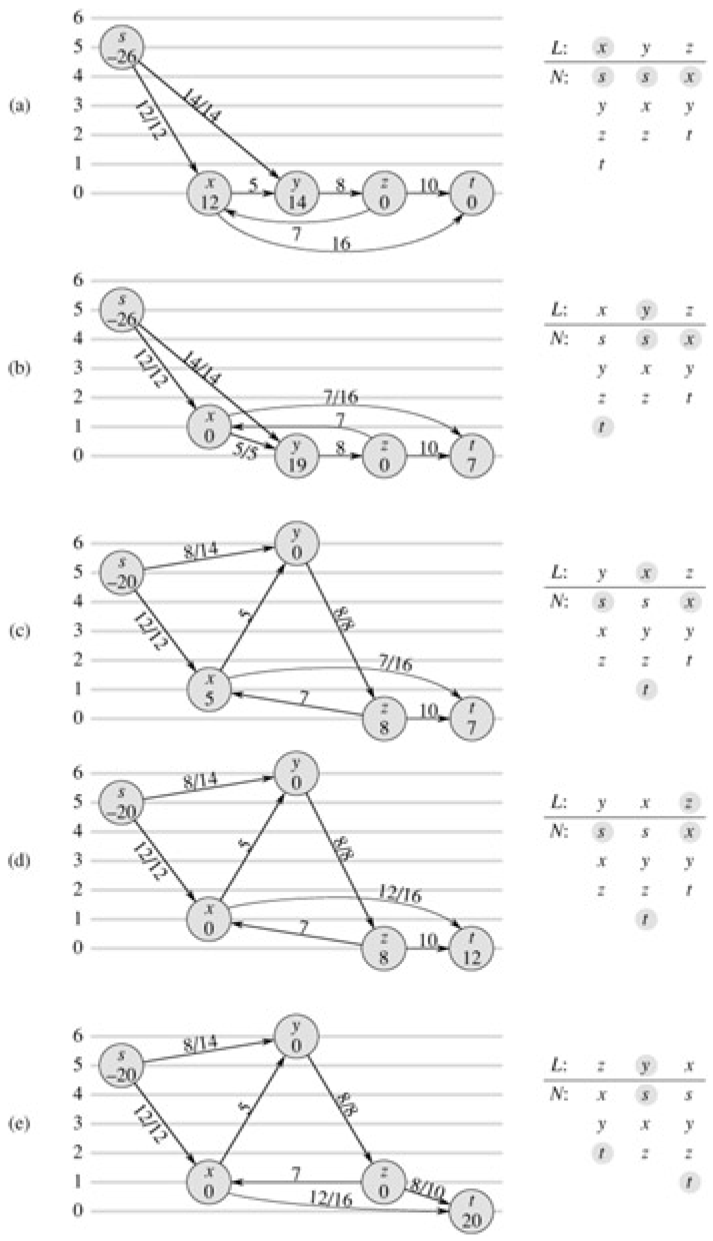

Figure ٢٦.١٠: The action of RELABEL-TO-FRONT. *(a)* A flow network just before the first iteration of the *while* loop. Initially, ٢٦ units of flow leave source *s*. On the right is shown the initial list *L* = 〉 *x, y, z*〉 , where initially *u* = *x*. Under each vertex in list *L* is its neighbor list, with the current neighbor shaded. Vertex *x* is discharged. It is relabeled to height ١, ٥ units of excess flow are pushed to *y*, and the ٧ remaining units of excess are pushed to the sink *t*. Because *x* is relabeled, it is moved to the head of *L*, which in this case does not change the structure of *L*. *(b)* After *x*, the next vertex in *L* that is discharged is *y*. Figure ٢٦.٩ 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 ٥ units of excess flow to *t*. Because vertex *x* is not relabeled in this discharge operation, it remains in place in list *L*. *(d)* Since vertex *z* follows vertex *x* in *L*, it is discharged. It is relabeled to height ١ and all ٨ units of excess flow are pushed to *t*. Because *z* is relabeled, it is moved to the front of *L*. *(e)* Vertex *y* now follows vertex *z* in *L* and is therefore discharged. But because *y* has no excess, DISCHARGE immediately returns, and *y* remains in place in *L*. Vertex *x* is then discharged. Because it, too, has no excess, DISCHARGE again returns, and *x* remains in place in *L*. RELABEL-TO-FRONT has reached the end of list *L* and terminates. There are no overflowing vertices, and the preflow is a maximum flow.

To show that RELABEL-TO-FRONT computes a maximum flow, we shall show that it is an implementation of the generic push-relabel algorithm. First, observe that it performs push and relabel operation only when they apply, since Lemma ٢٦.٣٠ guarantees that DISCHARGE only performs them when they apply. It remains to show that when RELABEL-TO-FRONT terminates, no basic operations apply. The remainder of the correctness argument relies on the following loop invariant:

- At each test in line ٦ of RELABEL-TO-FRONT, list *L* is a topological sort of the vertices in the admissible network $G_{f,h} = (V, E_{f,h})$, and no vertex before *u* in the list has excess flow.
- **Initialization:** Immediately after INITIALIZE-PREFLOW has been run, *h*[*s*] = |*V*| and *h*[*v*] = ٠ for all $v \in V$ - {*s*}. Since |*V* | = ٢ (because *V* contains at least *s* and *t*), no edge can be admissible. Thus, $E_{f,h}$ = ø, and any ordering of *V* - {*s*, *t*} is a topological sort of $G_{f,h}$.

  Since *u* is initially the head of the list *L*, there are no vertices before it and so there are none before it with excess flow.
- **Maintenance:** To see that the topological sort is maintained by each iteration of the **while** loop, we start by observing that the admissible network is changed only by push and relabel operations. By Lemma ٢٦.٢٨, push operations do not cause edges to become admissible. Thus, admissible edges can be created only by relabel operations. After a vertex *u* is relabeled, however, Lemma ٢٦.٢٩ 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 ١١) and either no other vertices (if *u* is relabeled) or the same vertices as before (if *u* is not relabeled). Since *u* is discharged, it has no excess flow afterward. Thus, if *u* is relabeled during the discharge, no vertices preceding *u*' have excess flow. If *u* is not relabeled during the discharge, no vertices before it on the list acquired excess flow during this discharge, because *L* remained topologically sorted at all times during the discharge (as pointed out just above, admissible edges are created only by relabeling, not pushing), and so each push operation causes excess flow to move only to vertices further down the list (or to *s* or *t*). Again, no vertices preceding *u*' have excess flow.
- **Termination:** When the loop terminates, *u* is just past the end of *L*, and so the loop invariant ensures that the excess of every vertex is ٠. Thus, no basic operations apply.

Analysis

We shall now show that RELABEL-TO-FRONT runs in $O(V^٣)$ 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 ٢٦.٢٢, which provides an *O*(*V*) bound on the number of relabel operations executed per vertex and an $O(V^٢)$ bound on the total number of relabel operations overall. In addition, Exercise ٢٦.٤-٢ provides an *O*(*VE*) bound on the total time spent performing relabel operations, and Lemma ٢٦.٢٣ provides an *O*(*VE*) bound on the total number of saturating push operations.

Theorem ٢٦.٣١

The running time of RELABEL-TO-FRONT on any flow network *G* = (*V*, *E*) is $O(V^٣)$.

Proof Let us consider a "phase" of the relabel-to-front algorithm to be the time between two consecutive relabel operations. There are $O(V^٢)$ phases, since there are $O(V^٢)$ relabel operations. Each phase consists of at most |*V*| calls to DISCHARGE, which can be seen as follows. If DISCHARGE does not perform a re-label 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 $O(V^٢)$ phases, the number of times DISCHARGE is called in line ٨ of RELABEL-TO-FRONT is $O(V^٣)$. Thus, the total work performed by the **while** loop in RELABEL-TO-FRONT, excluding the work performed within DISCHARGE, is at most $O(V^٣)$.

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 ٤–٥). Exercise ٢٦.٤-٢ provides an $O(VE)$ time bound on all the $O(V^{٢})$ relabels that are performed.

Now, suppose that the action updates the *current*[*u*] pointer in line ٨. 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.٤-١).

The third type of action performed by DISCHARGE is a push operation (line ٧). 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 ٠. Thus, there can be at most one nonsaturating push per call to DISCHARGE. As we have observed, DISCHARGE is called $O(V^{٣})$ times, and thus the total time spent performing nonsaturating pushes is $O(V^{٣})$.

The running time of RELABEL-TO-FRONT is therefore $O(V^{٣} + VE)$, which is $O(V^{٣})$.

Exercises ٢٦.٥-١

Illustrate the execution of RELABEL-TO-FRONT in the manner of Figure ٢٦.١٠ for the flow network in Figure ٢٦.١(a). Assume that the initial ordering of vertices in *L* is $\langle v_{١}, v_{٢}, v_{٣}, v_{٤}\rangle$ and that the neighbor lists are

$$\begin{aligned} N[v_{١}] &= \langle s, v_{٢}, v_{٣}\rangle , \\ N[v_{٢}] &= \langle s, v_{١}, v_{٣}, v_{٤}\rangle , \\ N[v_{٣}] &= \langle v_{١}, v_{٢}, v_{٤}, t\rangle , \\ N[v_{٤}] &= \langle v_{٢}, v_{٣}, t\rangle . \end{aligned}$$

Exercises ٢٦.٥.٢: ★

We would like to implement a push-relabel algorithm in which we maintain a first-in, first-out queue of overflowing vertices. The algorithm repeatedly discharges the vertex at the head of the queue, and any vertices that were not overflowing before the discharge but are overflowing afterward are placed at the end of the queue. After the vertex at the head of the queue is discharged, it is removed. When the queue is empty, the algorithm terminates. Show that this algorithm can be implemented to compute a maximum flow in $O(V^{٣})$ time.

Exercises ٢٦.٥-٣

Show that the generic algorithm still works if RELABEL updates $h[u]$ by simply computing $h[u] \leftarrow h[u] + ١$. How would this change affect the analysis of RELABEL-TO-FRONT?

Exercises ٢٦.٥-٤: ★

Show that if we always discharge a highest overflowing vertex, the push-relabel method can be made to run in $O(V^{٣})$ time.

Exercises ٢٦.٥-٥

Suppose that at some point in the execution of a push-relabel algorithm, there exists an integer ٠ < $k \le |V|$ - ١ for which no vertex has $h[v] = k$. Show that all vertices with $h[v] > k$ are on the source side of a minimum cut. If such a k exists, the ***gap heuristic*** updates every vertex $v \in V - s$ for which $h[v] > k$ to set $h[v] \leftarrow \max(h[v], |V|$+١). 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 ٢٦-١: Escape problem

An $n \times n$ ***grid*** is an undirected graph consisting of n rows and n columns of vertices, as shown in Figure ٢٦.١١. 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 = ١, $i = n$, j = ١, or $j = n$.

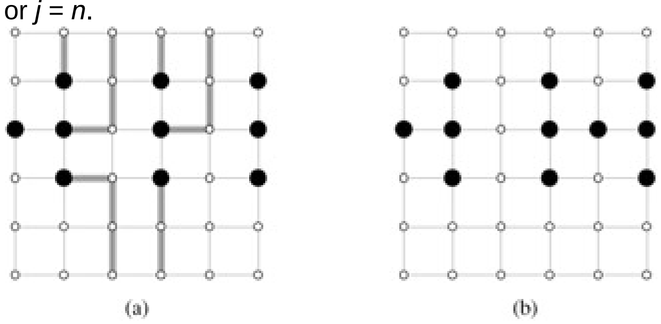

Figure ٢٦.١١: Grids for the escape problem. Starting points are black, and other grid vertices are white. *(a)* A grid with an escape, shown by shaded paths. *(b)* A grid with no escape.

Given $m \le n^{٢}$ starting points $(x_١, y_١), (x_٢, y_٢), . . . ,(x_m, y_m)$ 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 ٢٦.١١(a) has an escape, but the grid in Figure ٢٦.١١(b) does not.

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

Problems ٢٦-٢: 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 ٠. 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 = {١, ٢, . . . ,n}, construct the graph $G' = (V', E')$, where

$$V' = \{x_٠, x_١, . . . ,x_n\} \cup \{y_٠, y_١, . . . ,y_n\} ,$$

$$E' = \{(x_٠, x_i) : i \in V\} \cup \{(y_i, y_٠) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\} ,$$

b. and run a maximum-flow algorithm.)
c. Does your algorithm work for directed graphs that contain cycles? Explain.

Problems ٢٦-٣: Space shuttle experiments

Professor Spock is consulting for NASA, which is planning a series of space shuttle flights and must decide which commercial experiments to perform and which instruments to have on board each flight. For each flight, NASA considers a set $E = \{E_{١}, E_{٢}, . . . ,E_m\}$ of experiments, and the commercial sponsor of experiment E_j has agreed to pay NASA p_j dollars for the results of the experiment. The experiments use a set $I = \{I_{١}, I_{٢}, . . . ,I_n\}$ of instruments; each experiment E_j requires all the instruments in a subset $R_j \subseteq I$. The cost of carrying instrument I_k is c_k dollars. The professor's job is to find an efficient algorithm to determine which experiments to perform and which instruments to carry for a given flight in order to maximize the net revenue, which is the total income from experiments performed minus the total cost of all instruments carried.

Consider the following network G. The network contains a source vertex s, vertices $I_{١}, I_{٢}, . . . ,I_n$, vertices $E_{١}, E_{٢}, . . . ,E_m$, and a sink vertex t. For $k =$ ١, ٢. . . ,n, there is an edge (s, I_k) of capacity c_k, and for $j =$ ١, ٢,. . . ,m, there is an edge (E_j, t) of capacity p_j. For $k =$ ١, ٢, . . . ,n and $j =$ ١, ٢, . . . ,m, if $I_k \in R_j$, then there is an edge (I_k, E_j) of infinite capacity.

a. Show that if $E_j \in T$ for a finite-capacity cut (S, T) of G, then $I_k \in T$ for each $I_k \in R_j$.
b. Show how to determine the maximum net revenue from the capacity of the minimum cut of G and the given p_j values.
c. Give an efficient algorithm to determine which experiments to perform and which instruments to carry. Analyze the running time of your algorithm in terms of m, n, and $r = \sum_{j=1}^{m} |R_j|$.

Problem ٢٦-٤: Updating maximum flow

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

a. Suppose that the capacity of a single edge $(u, v) \in E$ is increased by ١. Give an $O(V + E)$-time algorithm to update the maximum flow.
b. Suppose that the capacity of a single edge $(u, v) \in E$ is decreased by ١. Give an $O(V + E)$-time algorithm to update the maximum flow.

Problem ٢٦-٥: Maximum flow by scaling

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

a. Argue that a minimum cut of G has capacity at most $C\,|E|$.
b. For a given number K, show that an augmenting path of capacity at least K can be found in $O(E)$ time, if such a path exists.

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

```
MAX-FLOW-BY-SCALING (G, s, t)
C ← max_(u.v)∈E c(u, v)
initialize flow f to ٠
K ← ٢^⌊lgC⌋
while K ≥ ١
    do while there exists an augmenting path p of capacity at least K
           do augment flow f along p
       K ← K/٢
return f
```

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

d. Show that the capacity of a minimum cut of the residual graph G_f is at most ٢*K* |*E*| each time line ٤ is executed.

e. Argue that the inner **while** loop of lines ٥–٦ is executed $O(E)$ times for each value of *K*.

f. Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs in $O(E^{٢} \lg C)$ time.

Problem ٢٦-٦: Maximum flow with negative capacities

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

a. Consider an edge (u, v) in a flow network $G = (V, E)$ with $c(u, v) <$ ٠. Briefly explain what such a negative capacity means in terms of the flow between *u* and *v*.

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

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

and

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

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

$c'(u, v) = c'(v, u) = (c(u, v) + c(v, u))/$٢ .

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

$c'(s', u) = \max($٠$, (c(V, u) - c(u, V))/$٢$)$

and

$c'(u, t') = \max($٠$, (c(u, V) - c(V, u))/$٢$)$.

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

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

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

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

Problem ٢٦-٧: 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 $|M \oplus P| = |M| +$ ١. Show that if $P_{١}, P_{٢}, . . . ,P_k$ are vertex-disjoint augmenting paths with respect to M, then the symmetric difference $M \oplus (P_{١} \cup P_{٢} \cup \cdot\cdot\cdot \cup P_k)$ is a matching with cardinality $|M| + k$.

The general structure of our algorithm is the following:

```
HOPCROFT-KARP (G)
١  $M \leftarrow \emptyset$
٢  **repeat**
٣      let $\mathcal{P} \leftarrow \{P_1, P_2, \ldots, P_k\}$ be a maximum set of vertex-disjoint
           shortest augmenting paths with respect to $M$
٤      $M \leftarrow M \oplus (P_{١} \cup P_{٢} \cup \cdot\cdot\cdot \cup P_k)$
٥    **until** $\mathcal{P} = \emptyset$
٦  **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 ٣.

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 ٢. Conclude that G' is a disjoint union of simple paths or cycles. Argue that edges in each such simple path or cycle belong alternately to M or M^*. Prove that if $|M| \le |M^*|$, then $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M.

Let l be the length of a shortest augmenting path with respect to a matching M, and let $P_{١}, P_{٢}, . . . ,P_k$ be a maximum set of vertex-disjoint augmenting paths of length l with respect to M. Let M' $= M \oplus (P_{١} \cup \cdot\cdot\cdot \cup P_k)$, and suppose that P is a shortest augmenting path with respect to M'.

c. Show that if P is vertex-disjoint from $P_{١}, P_{٢}, . . . ,P_k$, then P has more than l edges.
d. Now suppose that P is not vertex-disjoint from $P_{١}, P_{٢}, . . . ,P_k$. Let A be the set of edges $(M \oplus M') \oplus P$. Show that $A = (P_{١} \cup P_{٢} \cup \cdot\cdot\cdot \cup P_k) \oplus P$ and that $|A| \ge (k +$ ١$)l$. Conclude that P has more than l edges.
e. Prove that if a shortest augmenting path for M has length l, the size of the maximum matching is at most $|M| + |V| /l$.
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 maximum set of vertex-disjoint shortest augmenting paths $P_{١}, P_{٢}, . . . ,P_k$ for a given matching M. Conclude that the total running time of HOPCROFT-KARP is $O(\sqrt{V}E)$.