In the next section, we discuss how to select augmenting paths so as to avoid the potential bad behavior of the algorithm.

7.3 Choosing Good Augmenting Paths

In the previous section, we saw that any way of choosing an augmenting path increases the value of the flow, and this led to a bound of C on the number of augmentations, where $C = \sum_{e \text{ out of } s} c_e$. When C is not very large, this can be a reasonable bound; however, it is very weak when C is large.

To get a sense for how bad this bound can be, consider the example graph in Figure 7.2; but this time assume the capacities are as follows: The edges (s, v), (s, u), (v, t) and (u, t) have capacity 100, and the edge (u, v) has capacity 1, as shown in Figure 7.6. It is easy to see that the maximum flow has value 200, and has $f(e) = 100$ for the edges (s, v), (s, u), (v, t) and (u, t) and value 0 on the edge (u, v). This flow can be obtained by a sequence of two augmentations, using the paths of nodes s, u, t and path s, v, t. But consider how bad the Ford-Fulkerson Algorithm can be with pathological choices for the augmenting paths. Suppose we start with augmenting path P_1 of nodes s, u, v, t in this order (as shown in Figure 7.6). This path has $\texttt{bottleneck}(P_1, f) = 1$. After this augmentation, we have $f(e) = 1$ on the edge $e = (u, v)$, so the reverse edge is in the residual graph. For the next augmenting path, we choose the path P_2 of the nodes s, v, u, t in this order. In this second augmentation, we get $\texttt{bottleneck}(P_2, f) = 1$ as well. After this second augmentation, we have $f(e) = 0$ for the edge $e = (u, v)$, so the edge is again in the residual graph. Suppose we alternate between choosing P_1 and P_2 for augmentation. In this case, each augmentation will have 1 as the bottleneck capacity, and it will take 200 augmentations to get the desired flow of value 200. This is exactly the bound we proved in (7.4), since $C = 200$ in this example.

Designing a Faster Flow Algorithm

The goal of this section is to show that with a better choice of paths, we can improve this bound significantly. A large amount of work has been devoted to finding good ways of choosing augmenting paths in the Maximum-Flow Problem so as to minimize the number of iterations. We focus here on one of the most natural approaches and will mention other approaches at the end of the section. Recall that augmentation increases the value of the maximum flow by the bottleneck capacity of the selected path; so if we choose paths with large bottleneck capacity, we will be making a lot of progress. A natural idea is to select the path that has the largest bottleneck capacity. Having to find such paths can slow down each individual iteration by quite a bit. We will avoid this slowdown by not worrying about selecting the path that has *exactly*

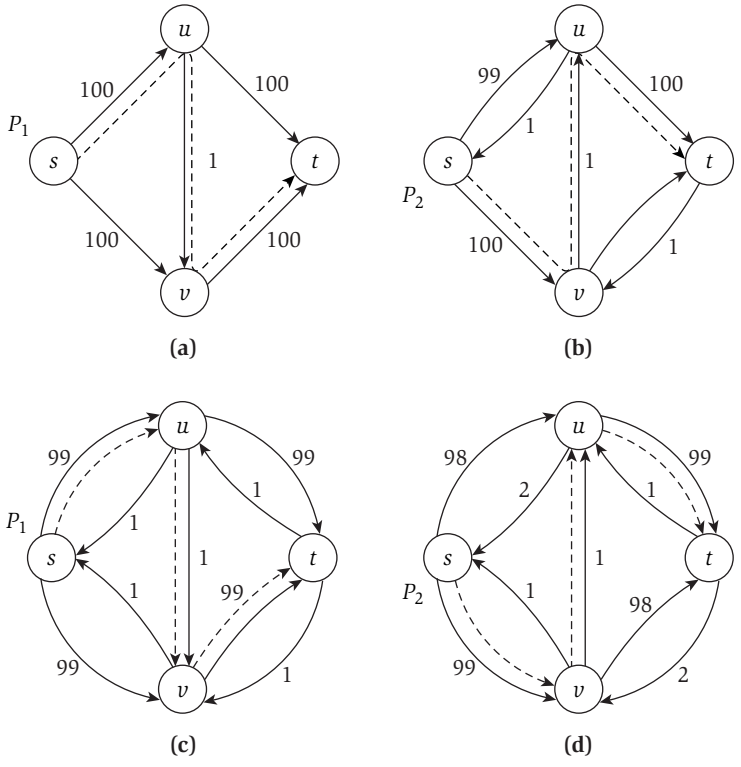

Figure 7.6 Parts (a) through (d) depict four iterations of the Ford-Fulkerson Algorithm using a bad choice of augmenting paths: The augmentations alternate between the path P_1 through the nodes s, u, v, t in order and the path P_2 through the nodes s, v, u, t in order.

the largest bottleneck capacity. Instead, we will maintain a so-called *scaling parameter* Δ, and we will look for paths that have bottleneck capacity of at least Δ.

Let $G_f(\Delta)$ be the subset of the residual graph consisting only of edges with residual capacity of at least Δ. We will work with values of Δ that are powers of 2. The algorithm is as follows.

```
Scaling Max-Flow
  Initially f(e) = 0 for all e in G
  Initially set Δ to be the largest power of 2 that is no larger
        than the maximum capacity out of s: Δ ≤ max_{e out of s} c_e
    While Δ ≥ 1
       While there is an s-t path in the graph G_f(Δ)
          Let P be a simple s-t path in G_f(Δ)
```

```
            f' = augment(f, P)
            Update f to be f' and update G_f(Δ)
        Endwhile
        Δ = Δ/2
    Endwhile
Return f
```

Analyzing the Algorithm

First observe that the new Scaling Max-Flow Algorithm is really just an implementation of the original Ford-Fulkerson Algorithm. The new loops, the value Δ, and the restricted residual graph $G_f(\Delta)$ are only used to guide the selection of residual path—with the goal of using edges with large residual capacity for as long as possible. Hence all the properties that we proved about the original Max-Flow Algorithm are also true for this new version: the flow remains integer-valued throughout the algorithm, and hence all residual capacities are integer-valued.

(7.15) *If the capacities are integer-valued, then throughout the Scaling Max-Flow Algorithm the flow and the residual capacities remain integer-valued. This implies that when $\Delta = 1$, $G_f(\Delta)$ is the same as G_f, and hence when the algorithm terminates the flow, f is of maximum value.*

Next we consider the running time. We call an iteration of the outside `While` loop—with a fixed value of Δ—the Δ-*scaling phase*. It is easy to give an upper bound on the number of different Δ-scaling phases, in terms of the value $C = \sum_{e \text{ out of } s} c_e$ that we also used in the previous section. The initial value of Δ is at most C, it drops by factors of 2, and it never gets below 1. Thus,

(7.16) *The number of iterations of the outer* `While` *loop is at most* $1 + \lceil \log_2 C \rceil$.

The harder part is to bound the number of augmentations done in each scaling phase. The idea here is that we are using paths that augment the flow by a lot, and so there should be relatively few augmentations. During the Δ-scaling phase, we only use edges with residual capacity of at least Δ. Using (7.3), we have

(7.17) *During the Δ-scaling phase, each augmentation increases the flow value by at least Δ.*

The key insight is that at the end of the Δ-scaling phase, the flow f cannot be too far from the maximum possible value.

(7.18) *Let f be the flow at the end of the Δ-scaling phase. There is an s-t cut (A, B) in G for which $c(A, B) \leq v(f) + m\Delta$, where m is the number of edges in the graph G. Consequently, the maximum flow in the network has value at most $v(f) + m\Delta$.*

Proof. This proof is analogous to our proof of (7.9), which established that the flow returned by the original Max-Flow Algorithm is of maximum value.

As in that proof, we must identify a cut (A, B) with the desired property. Let A denote the set of all nodes v in G for which there is an s-v path in $G_f(\Delta)$. Let B denote the set of all other nodes: $B = V - A$. We can see that (A, B) is indeed an s-t cut as otherwise the phase would not have ended.

Now consider an edge $e = (u, v)$ in G for which $u \in A$ and $v \in B$. We claim that $c_e < f(e) + \Delta$. For if this were not the case, then e would be a forward edge in the graph $G_f(\Delta)$, and since $u \in A$, there is an s-u path in $G_f(\Delta)$; appending e to this path, we would obtain an s-v path in $G_f(\Delta)$, contradicting our assumption that $v \in B$. Similarly, we claim that for any edge $e' = (u', v')$ in G for which $u' \in B$ and $v' \in A$, we have $f(e') < \Delta$. Indeed, if $f(e') \geq \Delta$, then e' would give rise to a backward edge $e'' = (v', u')$ in the graph $G_f(\Delta)$, and since $v' \in A$, there is an s-v' path in $G_f(\Delta)$; appending e'' to this path, we would obtain an s-u' path in $G_f(\Delta)$, contradicting our assumption that $u' \in B$.

So all edges e out of A are almost saturated—they satisfy $c_e < f(e) + \Delta$—and all edges into A are almost empty—they satisfy $f(e) < \Delta$. We can now use (7.6) to reach the desired conclusion:

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) \\ &\geq \sum_{e \text{ out of } A} (c_e - \Delta) - \sum_{e \text{ into } A} \Delta \\ &= \sum_{e \text{ out of } A} c_e - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ into } A} \Delta \\ &\geq c(A, B) - m\Delta. \end{aligned}$$

Here the first inequality follows from our bounds on the flow values of edges across the cut, and the second inequality follows from the simple fact that the graph only contains m edges total.

The maximum-flow value is bounded by the capacity of any cut by (7.8). We use the cut (A, B) to obtain the bound claimed in the second statement.

■

(7.19) *The number of augmentations in a scaling phase is at most $2m$.*

Proof. The statement is clearly true in the first scaling phase: we can use each of the edges out of s only for at most one augmentation in that phase. Now consider a later scaling phase Δ, and let f_p be the flow at the end of the *previous* scaling phase. In that phase, we used $\Delta' = 2\Delta$ as our parameter. By (7.18), the maximum flow f^* is at most $v(f^*) \leq v(f_p) + m\Delta' = v(f_p) + 2m\Delta$. In the Δ-scaling phase, each augmentation increases the flow by at least Δ, and hence there can be at most $2m$ augmentations. ■

An augmentation takes $O(m)$ time, including the time required to set up the graph and find the appropriate path. We have at most $1 + \lceil \log_2 C \rceil$ scaling phases and at most $2m$ augmentations in each scaling phase. Thus we have the following result.

(7.20) *The Scaling Max-Flow Algorithm in a graph with m edges and integer capacities finds a maximum flow in at most $2m(1 + \lceil \log_2 C \rceil)$ augmentations. It can be implemented to run in at most $O(m^2 \log_2 C)$ time.*

When C is large, this time bound is much better than the $O(mC)$ bound that applied to an arbitrary implementation of the Ford-Fulkerson Algorithm. In our example at the beginning of this section, we had capacities of size 100, but we could just as well have used capacities of size 2^{100}; in this case, the generic Ford-Fulkerson Algorithm could take time proportional to 2^{100}, while the scaling algorithm will take time proportional to $\log_2(2^{100}) = 100$. One way to view this distinction is as follows: The generic Ford-Fulkerson Algorithm requires time proportional to the *magnitude* of the capacities, while the scaling algorithm only requires time proportional to the number of *bits* needed to specify the capacities in the input to the problem. As a result, the scaling algorithm is running in time polynomial in the size of the input (i.e., the number of edges and the numerical representation of the capacities), and so it meets our traditional goal of achieving a polynomial-time algorithm. Bad implementations of the Ford-Fulkerson Algorithm, which can require close to C iterations, do not meet this standard of polynomiality. (Recall that in Section 6.4 we used the term *pseudo-polynomial* to describe such algorithms, which are polynomial in the magnitudes of the input numbers but not in the number of bits needed to represent them.)

Extensions: Strongly Polynomial Algorithms

Could we ask for something qualitatively better than what the scaling algorithm guarantees? Here is one thing we could hope for: Our example graph (Figure 7.6) had four nodes and five edges; so it would be nice to use a

number of iterations that is polynomial in the numbers 4 and 5, completely independently of the values of the capacities. Such an algorithm, which is polynomial in $|V|$ and $|E|$ only, and works with numbers having a polynomial number of bits, is called a *strongly polynomial algorithm*. In fact, there is a simple and natural implementation of the Ford-Fulkerson Algorithm that leads to such a strongly polynomial bound: each iteration chooses the augmenting path with the fewest number of edges. Dinitz, and independently Edmonds and Karp, proved that with this choice the algorithm terminates in at most $O(mn)$ iterations. In fact, these were the first polynomial algorithms for the Maximum-Flow Problem. There has since been a huge amount of work devoted to improving the running times of maximum-flow algorithms. There are currently algorithms that achieve running times of $O(mn \log n)$, $O(n^3)$, and $O(\min(n^{2/3}, m^{1/2})m \log n \log U)$, where the last bound assumes that all capacities are integral and at most U. In the next section, we'll discuss a strongly polynomial maximum-flow algorithm based on a different principle.

* 7.4 The Preflow-Push Maximum-Flow Algorithm

From the very beginning, our discussion of the Maximum-Flow Problem has been centered around the idea of an augmenting path in the residual graph. However, there are some very powerful techniques for maximum flow that are not explicitly based on augmenting paths. In this section we study one such technique, the Preflow-Push Algorithm.

Designing the Algorithm

Algorithms based on augmenting paths maintain a flow f, and use the `augment` procedure to increase the value of the flow. By way of contrast, the Preflow-Push Algorithm will, in essence, increase the flow on an edge-by-edge basis. Changing the flow on a single edge will typically violate the conservation condition, and so the algorithm will have to maintain something less well behaved than a flow—something that does not obey conservation—as it operates.

Preflows We say that an *s-t preflow* (*preflow*, for short) is a function f that maps each edge e to a nonnegative real number, $f : E \to \mathbf{R}^+$. A preflow f must satisfy the capacity conditions:

(i) For each $e \in E$, we have $0 \leq f(e) \leq c_e$.

In place of the conservation conditions, we require only inequalities: Each node other than s must have at least as much flow entering as leaving.

(ii) For each node v other than the source s, we have

$$\sum_{e \text{ into } v} f(e) \geq \sum_{e \text{ out of } v} f(e).$$

We will call the difference

$$e_f(v) = \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e)$$

the *excess* of the preflow at node v. Notice that a preflow where all nodes other than s and t have zero excess is a flow, and the value of the flow is exactly $e_f(t) = -e_f(s)$. We can still define the concept of a residual graph G_f for a preflow f, just as we did for a flow. The algorithm will "push" flow along edges of the residual graph (using both forward and backward edges).

Preflows and Labelings The Preflow-Push Algorithm will maintain a preflow and work on converting the preflow into a flow. The algorithm is based on the physical intuition that flow naturally finds its way "downhill." The "heights" for this intuition will be labels $h(v)$ for each node v that the algorithm will define and maintain, as shown in Figure 7.7. We will push flow from nodes with higher labels to those with lower labels, following the intuition that fluid flows downhill. To make this precise, a *labeling* is a function $h : V \to \mathbf{Z}_{\geq 0}$ from the nodes to the nonnegative integers. We will also refer to the labels as *heights* of the nodes. We will say that a labeling h and an s-t preflow f are *compatible* if

(i) (*Source and sink conditions*) $h(t) = 0$ and $h(s) = n$,

(ii) (*Steepness conditions*) For all edges $(v, w) \in E_f$ in the residual graph, we have $h(v) \leq h(w) + 1$.

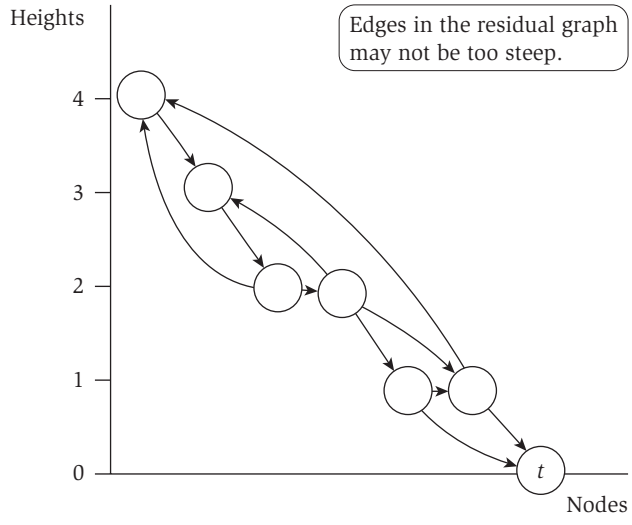

Figure 7.7 A residual graph and a compatible labeling. No edge in the residual graph can be too "steep"—its tail can be at most one unit above its head in height. The source node s must have $h(s) = n$ and is not drawn in the figure.

Intuitively, the height difference n between the source and the sink is meant to ensure that the flow starts high enough to flow from s toward the sink t, while the steepness condition will help by making the descent of the flow gradual enough to make it to the sink.

The key property of a compatible preflow and labeling is that there can be no s-t path in the residual graph.

(7.21) *If s-t preflow f is compatible with a labeling h, then there is no s-t path in the residual graph G_f.*

Proof. We prove the statement by contradiction. Let P be a simple s-t path in the residual graph G. Assume that the nodes along P are $s, v_1, \ldots, v_k = t$. By definition of a labeling compatible with preflow f, we have that $h(s) = n$. The edge (s, v_1) is in the residual graph, and hence $h(v_1) \geq h(s) - 1 = n - 1$. Using induction on i and the steepness condition for the edge (v_{i-1}, v_i), we get that for all nodes v_i in path P the height is at least $h(v_i) \geq n - i$. Notice that the last node of the path is $v_k = t$; hence we get that $h(t) \geq n - k$. However, $h(t) = 0$ by definition; and $k < n$ as the path P is simple. This contradiction proves the claim. ■

Recall from (7.9) that if there is no s-t path in the residual graph G_f of a *flow* f, then the flow has maximum value. This implies the following corollary.

(7.22) *If s-t flow f is compatible with a labeling h, then f is a flow of maximum value.*

Note that (7.21) applies to preflows, while (7.22) is more restrictive in that it applies only to flows. Thus the Preflow-Push Algorithm will maintain a preflow f and a labeling h compatible with f, and it will work on modifying f and h so as to move f toward being a flow. Once f actually becomes a flow, we can invoke (7.22) to conclude that it is a maximum flow. In light of this, we can view the Preflow-Push Algorithm as being in a way orthogonal to the Ford-Fulkerson Algorithm. The Ford-Fulkerson Algorithm maintains a feasible flow while changing it gradually toward optimality. The Preflow-Push Algorithm, on the other hand, maintains a condition that would imply the optimality of a preflow f, *if it were to be a feasible flow*, and the algorithm gradually transforms the preflow f into a flow.

To start the algorithm, we will need to define an initial preflow f and labeling h that are compatible. We will use $h(v) = 0$ for all $v \neq s$, and $h(s) = n$, as our initial labeling. To make a preflow f compatible with this labeling, we need to make sure that no edges leaving s are in the residual graph (as these edges do not satisfy the steepness condition). To this end, we define the initial

preflow as $f(e) = c_e$ for all edges $e = (s, v)$ leaving the source, and $f(e) = 0$ for all other edges.

(7.23) *The initial preflow f and labeling h are compatible.*

Pushing and Relabeling Next we will discuss the steps the algorithm makes toward turning the preflow f into a feasible flow, while keeping it compatible with some labeling h. Consider any node v that has excess—that is, $e_f(v) > 0$. If there is any edge e in the residual graph G_f that leaves v and goes to a node w at a lower height (note that $h(w)$ is at most 1 less than $h(v)$ due to the steepness condition), then we can modify f by pushing some of the excess flow from v to w. We will call this a *push* operation.

```
push(f, h, v, w)
  Applicable if e_f(v) > 0, h(w) < h(v) and (v, w) ∈ E_f
  If e = (v, w) is a forward edge then
    let δ = min(e_f(v), c_e − f(e)) and
    increase f(e) by δ
  If (v, w) is a backward edge then
    let e = (w, v), δ = min(e_f(v), f(e)) and
    decrease f(e) by δ
  Return(f, h)
```

If we cannot push the excess of v along any edge leaving v, then we will need to raise v's height. We will call this a *relabel* operation.

```
relabel(f, h, v)
  Applicable if e_f(v) > 0, and
     for all edges (v, w) ∈ E_f we have h(w) ≥ h(v)
  Increase h(v) by 1
  Return(f, h)
```

The Full Preflow-Push Algorithm So, in summary, the Preflow-Push Algorithm is as follows.

```
Preflow-Push
  Initially h(v) = 0 for all v ≠ s and h(s) = n and
  f(e) = c_e for all e = (s, v) and f(e) = 0 for all other edges
  While there is a node v ≠ t with excess e_f(v) > 0
    Let v be a node with excess
    If there is w such that push(f, h, v, w) can be applied then
       push(f, h, v, w)
```

```
  Else
     relabel(f, h, v)
Endwhile
Return(f)
```

Analyzing the Algorithm

As usual, this algorithm is somewhat underspecified. For an implementation of the algorithm, we will have to specify which node with excess to choose, and how to efficiently select an edge on which to push. However, it is clear that each iteration of this algorithm can be implemented in polynomial time. (We'll discuss later how to implement it reasonably efficiently.) Further, it is not hard to see that the preflow f and the labeling h are compatible throughout the algorithm. If the algorithm terminates—something that is far from obvious based on its description—then there are no nodes other than t with positive excess, and hence the preflow f is in fact a flow. It then follows from (7.22) that f would be a maximum flow at termination.

We summarize a few simple observations about the algorithm.

(7.24) *Throughout the Preflow-Push Algorithm:*

(i) the labels are nonnegative integers;

(ii) f is a preflow, and if the capacities are integral, then the preflow f is integral; and

(iii) the preflow f and labeling h are compatible.

If the algorithm returns a preflow f, then f is a flow of maximum value.

Proof. By (7.23) the initial preflow f and labeling h are compatible. We will show using induction on the number of `push` and `relabel` operations that f and h satisfy the properties of the statement. The `push` operation modifies the preflow f, but the bounds on δ guarantee that the f returned satisfies the capacity constraints, and that excesses all remain nonnegative, so f is a preflow. To see that the preflow f and the labeling h are compatible, note that `push`(f, h, v, w) can add one edge to the residual graph, the reverse edge (v, w), and this edge does satisfy the steepness condition. The `relabel` operation increases the label of v, and hence increases the steepness of all edges leaving v. However, it only applies when no edge leaving v in the residual graph is going downward, and hence the preflow f and the labeling h are compatible after relabeling.

The algorithm terminates if no node other than s or t has excess. In this case, f is a flow by definition; and since the preflow f and the labeling h

remain compatible throughout the algorithm, (7.22) implies that f is a flow of maximum value. ■

Next we will consider the number of `push` and `relabel` operations. First we will prove a limit on the `relabel` operations, and this will help prove a limit on the maximum number of `push` operations possible. The algorithm never changes the label of s (as the source never has positive excess). Each other node v starts with $h(v) = 0$, and its label increases by 1 every time it changes. So we simply need to give a limit on how high a label can get. We only consider a node v for `relabel` when v has excess. The only source of flow in the network is the source s; hence, intuitively, the excess at v must have originated at s. The following consequence of this fact will be key to bounding the labels.

(7.25) *Let f be a preflow. If the node v has excess, then there is a path in G_f from v to the source s.*

Proof. Let A denote all the nodes w such that there is a path from w to s in the residual graph G_f, and let $B = V - A$. We need to show that all nodes with excess are in A.

Notice that $s \in A$. Further, no edges $e = (x, y)$ leaving A can have positive flow, as an edge with $f(e) > 0$ would give rise to a reverse edge (y, x) in the residual graph, and then y would have been in A. Now consider the sum of excesses in the set B, and recall that each node in B has nonnegative excess, as $s \notin B$.

$$0 \leq \sum_{v \in B} e_f(v) = \sum_{v \in B} (f^{\text{in}}(v) - f^{\text{out}}(v))$$

Let's rewrite the sum on the right as follows. If an edge e has both ends in B, then $f(e)$ appears once in the sum with a "$+$" and once with a "$-$", and hence these two terms cancel out. If e has only its head in B, then e leaves A, and we saw above that all edges leaving A have $f(e) = 0$. If e has only its tail in B, then $f(e)$ appears just once in the sum, with a "$-$". So we get

$$0 \leq \sum_{v \in B} e_f(v) = -f^{\text{out}}(B).$$

Since flows are nonnegative, we see that the sum of the excesses in B is zero; since each individual excess in B is nonnegative, they must therefore all be 0. ■

Now we are ready to prove that the labels do not change too much. Recall that n denotes the number of nodes in V.

(7.26) *Throughout the algorithm, all nodes have* $h(v) \leq 2n - 1$.

Proof. The initial labels $h(t) = 0$ and $h(s) = n$ do not change during the algorithm. Consider some other node $v \neq s, t$. The algorithm changes v's label only when applying the `relabel` operation, so let f and h be the preflow and labeling returned by a `relabel`(f, h, v) operation. By (7.25) there is a path P in the residual graph G_f from v to s. Let $|P|$ denote the number of edges in P, and note that $|P| \leq n - 1$. The steepness condition implies that heights of the nodes can decrease by at most 1 along each edge in P, and hence $h(v) - h(s) \leq |P|$, which proves the statement. ■

Labels are monotone increasing throughout the algorithm, so this statement immediately implies a limit on the number of relabeling operations.

(7.27) *Throughout the algorithm, each node is relabeled at most* $2n - 1$ *times, and the total number of relabeling operations is less than* $2n^2$.

Next we will bound the number of `push` operations. We will distinguish two kinds of `push` operations. A `push`(f, h, v, w) operation is *saturating* if either $e = (v, w)$ is a forward edge in E_f and $\delta = c_e - f(e)$, or (v, w) is a backward edge with $e = (w, v)$ and $\delta = f(e)$. In other words, the push is saturating if, after the push, the edge (v, w) is no longer in the residual graph. All other `push` operations will be referred to as *nonsaturating*.

(7.28) *Throughout the algorithm, the number of saturating* `push` *operations is at most* $2nm$.

Proof. Consider an edge (v, w) in the residual graph. After a saturating `push`(f, h, v, w) operation, we have $h(v) = h(w) + 1$, and the edge (v, w) is no longer in the residual graph G_f, as shown in Figure 7.8. Before we can push again along this edge, first we have to push from w to v to make the edge (v, w) appear in the residual graph. However, in order to push from w to v, we first need for w's label to increase by at least 2 (so that w is above v). The label of w can increase by 2 at most $n - 1$ times, so a saturating push from v to w can occur at most n times. Each edge $e \in E$ can give rise to two edges in the residual graph, so overall we can have at most $2nm$ saturating pushes. ■

The hardest part of the analysis is proving a bound on the number of nonsaturating pushes, and this also will be the bottleneck for the theoretical bound on the running time.

(7.29) *Throughout the algorithm, the number of nonsaturating* `push` *operations is at most* $2n^2m$.

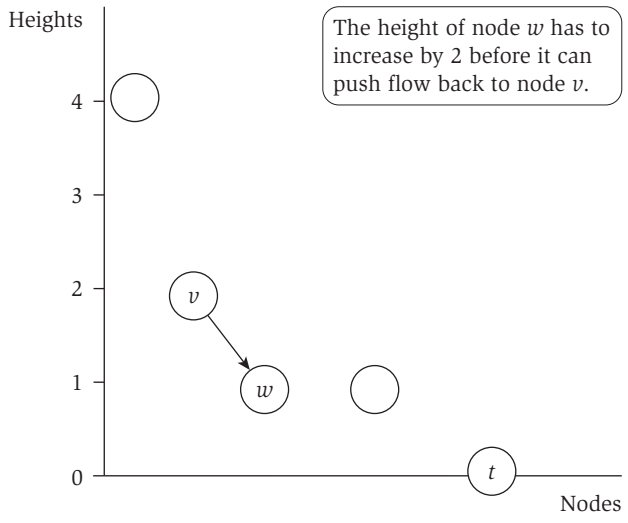

Figure 7.8 After a saturating push(f, h, v, w), the height of v exceeds the height of w by 1.

Proof. For this proof, we will use a so-called *potential function method*. For a preflow f and a compatible labeling h, we define

$$\Phi(f, h) = \sum_{v:e_f(v)>0} h(v)$$

to be the sum of the heights of all nodes with positive excess. (Φ is often called a *potential* since it resembles the "potential energy" of all nodes with positive excess.)

In the initial preflow and labeling, all nodes with positive excess are at height 0, so $\Phi(f, h) = 0$. $\Phi(f, h)$ remains nonnegative throughout the algorithm. A nonsaturating `push`(f, h, v, w) operation decreases $\Phi(f, h)$ by at least 1, since after the push the node v will have no excess, and w, the only node that gets new excess from the operation, is at a height 1 less than v. However, each saturating `push` and each `relabel` operation can increase $\Phi(f, h)$. A `relabel` operation increases $\Phi(f, h)$ by exactly 1. There are at most $2n^2$ `relabel` operations, so the total increase in $\Phi(f, h)$ due to `relabel` operations is $2n^2$. A saturating `push`(f, h, v, w) operation does not change labels, but it can increase $\Phi(f, h)$, since the node w may suddenly acquire positive excess after the push. This would increase $\Phi(f, h)$ by the height of w, which is at most $2n - 1$. There are at most $2nm$ saturating `push` operations, so the total increase in $\Phi(f, h)$ due to `push` operations is at most $2mn(2n - 1)$. So, between the two causes, $\Phi(f, h)$ can increase by at most $4mn^2$ during the algorithm.

But since Φ remains nonnegative throughout, and it decreases by at least 1 on each nonsaturating `push` operation, it follows that there can be at most $4mn^2$ nonsaturating `push` operations. ■

Extensions: An Improved Version of the Algorithm

There has been a lot of work devoted to choosing node selection rules for the Preflow-Push Algorithm to improve the worst-case running time. Here we consider a simple rule that leads to an improved $O(n^3)$ bound on the number of nonsaturating `push` operations.

(7.30) *If at each step we choose the node with excess at maximum height, then the number of nonsaturating* `push` *operations throughout the algorithm is at most* $4n^3$.

Proof. Consider the maximum height $H = \max_{v:e_f(v)>0} h(v)$ of any node with excess as the algorithm proceeds. The analysis will use this maximum height H in place of the potential function Φ in the previous $O(n^2m)$ bound.

This maximum height H can only increase due to relabeling (as flow is always pushed to nodes at lower height), and so the total increase in H throughout the algorithm is at most $2n^2$ by (7.26). H starts out 0 and remains nonnegative, so the number of times H changes is at most $4n^2$.

Now consider the behavior of the algorithm over a phase of time in which H remains constant. We claim that each node can have at most one nonsaturating `push` operation during this phase. Indeed, during this phase, flow is being pushed from nodes at height H to nodes at height $H-1$; and after a nonsaturating `push` operation from v, it must receive flow from a node at height $H+1$ before we can push from it again.

Since there are at most n nonsaturating `push` operations between each change to H, and H changes at most $4n^2$ times, the total number of nonsaturating `push` operations is at most $4n^3$. ■

As a follow-up to (7.30), it is interesting to note that experimentally the computational bottleneck of the method is the number of relabeling operations, and a better experimental running time is obtained by variants that work on increasing labels faster than one by one. This is a point that we pursue further in some of the exercises.

Implementing the Preflow-Push Algorithm

Finally, we need to briefly discuss how to implement this algorithm efficiently. Maintaining a few simple data structures will allow us to effectively implement

the operations of the algorithm in constant time each, and overall to implement the algorithm in time $O(mn)$ plus the number of nonsaturating `push` operations. Hence the generic algorithm will run in $O(mn^2)$ time, while the version that always selects the node at maximum height will run in $O(n^3)$ time.

We can maintain all nodes with excess on a simple list, and so we will be able to select a node with excess in constant time. One has to be a bit more careful to be able to select a node with maximum height H in constant time. In order to do this, we will maintain a linked list of all nodes with excess at every possible height. Note that whenever a node v gets relabeled, or continues to have positive excess after a `push`, it remains a node with maximum height H. Thus we only have to select a new node after a `push` when the current node v no longer has positive excess. If node v was at height H, then the new node at maximum height will also be at height H or, if no node at height H has excess, then the maximum height will be $H - 1$, since the previous `push` operation out of v pushed flow to a node at height $H - 1$.

Now assume we have selected a node v, and we need to select an edge (v, w) on which to apply `push`(f, h, v, w) (or `relabel`(f, h, v) if no such w exists). To be able to select an edge quickly, we will use the adjacency list representation of the graph. More precisely, we will maintain, for each node v, all possible edges leaving v in the residual graph (both forward and backward edges) in a linked list, and with each edge we keep its capacity and flow value. Note that this way we have two copies of each edge in our data structure: a forward and a backward copy. These two copies will have pointers to each other, so that updates done at one copy can be carried over to the other one in $O(1)$ time. We will select edges leaving a node v for `push` operations in the order they appear on node v's list. To facilitate this selection, we will maintain a pointer `current`(v) for each node v to the last edge on the list that has been considered for a `push` operation. So, if node v no longer has excess after a nonsaturating `push` operation out of node v, the pointer `current`(v) will stay at this edge, and we will use the same edge for the next `push` operation out of v. After a saturating `push` operation out of node v, we advance `current`(v) to the next edge on the list.

The key observation is that, after advancing the pointer `current`(v) from an edge (v, w), we will not want to apply `push` to this edge again until we relabel v.

(7.31) *After the* `current`(v) *pointer is advanced from an edge* (v, w)*, we cannot apply* `push` *to this edge until* v *gets relabeled.*

Proof. At the moment `current`(v) is advanced from the edge (v, w), there is some reason `push` cannot be applied to this edge. Either $h(w) \geq h(v)$, or the

edge is not in the residual graph. In the first case, we clearly need to relabel v before applying a `push` on this edge. In the latter case, one needs to apply `push` to the reverse edge (w, v) to make (v, w) reenter the residual graph. However, when we apply `push` to edge (w, v), then w is above v, and so v needs to be relabeled before one can push flow from v to w again. ■

Since edges do not have to be considered again for `push` before relabeling, we get the following.

(7.32) *When the* `current`(v) *pointer reaches the end of the edge list for* v*, the* `relabel` *operation can be applied to node* v*.*

After relabeling node v, we reset `current`(v) to the first edge on the list and start considering edges again in the order they appear on v's list.

(7.33) *The running time of the Preflow-Push Algorithm, implemented using the above data structures, is* $O(mn)$ *plus* $O(1)$ *for each nonsaturating* `push` *operation. In particular, the generic Preflow-Push Algorithm runs in* $O(n^2m)$ *time, while the version where we always select the node at maximum height runs in* $O(n^3)$ *time.*

Proof. The initial flow and relabeling is set up in $O(m)$ time. Both `push` and `relabel` operations can be implemented in $O(1)$ time, once the operation has been selected. Consider a node v. We know that v can be relabeled at most $2n$ times throughout the algorithm. We will consider the total time the algorithm spends on finding the right edge on which to `push` flow out of node v, between two times that node v gets relabeled. If node v has d_v adjacent edges, then by (7.32) we spend $O(d_v)$ time on advancing the `current`(v) pointer between consecutive relabelings of v. Thus the total time spent on advancing the `current` pointers throughout the algorithm is $O(\sum_{v \in V} nd_v) = O(mn)$, as claimed. ■

7.5 A First Application: The Bipartite Matching Problem

Having developed a set of powerful algorithms for the Maximum-Flow Problem, we now turn to the task of developing applications of maximum flows and minimum cuts in graphs. We begin with two very basic applications. First, in this section, we discuss the Bipartite Matching Problem mentioned at the beginning of this chapter. In the next section, we discuss the more general *Disjoint Paths Problem*.

The Problem

One of our original goals in developing the Maximum-Flow Problem was to be able to solve the Bipartite Matching Problem, and we now show how to do this. Recall that a *bipartite graph* $G = (V, E)$ is an undirected graph whose node set can be partitioned as $V = X \cup Y$, with the property that every edge $e \in E$ has one end in X and the other end in Y. A *matching* M in G is a subset of the edges $M \subseteq E$ such that each node appears in at most one edge in M. The Bipartite Matching Problem is that of finding a matching in G of largest possible size.

Designing the Algorithm

The graph defining a matching problem is undirected, while flow networks are directed; but it is actually not difficult to use an algorithm for the Maximum-Flow Problem to find a maximum matching.

Beginning with the graph G in an instance of the Bipartite Matching Problem, we construct a flow network G' as shown in Figure 7.9. First we direct all edges in G from X to Y. We then add a node s, and an edge (s, x) from s to each node in X. We add a node t, and an edge (y, t) from each node in Y to t. Finally, we give each edge in G' a capacity of 1.

We now compute a maximum s-t flow in this network G'. We will discover that the value of this maximum is equal to the size of the maximum matching in G. Moreover, our analysis will show how one can use the flow itself to recover the matching.

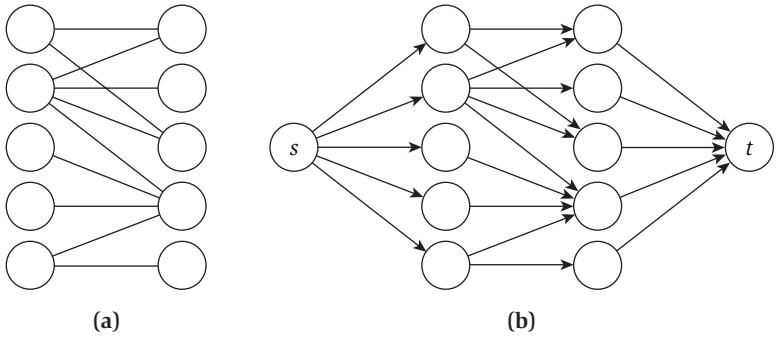

Figure 7.9 (a) A bipartite graph. (b) The corresponding flow network, with all capacities equal to 1.

Analyzing the Algorithm

The analysis is based on showing that integer-valued flows in G' encode matchings in G in a fairly transparent fashion. First, suppose there is a matching in G consisting of k edges $(x_{i_1}, y_{i_1}), \ldots, (x_{i_k}, y_{i_k})$. Then consider the flow f that sends one unit along each path of the form s, x_{i_j}, y_{i_j}, t—that is, $f(e) = 1$ for each edge on one of these paths. One can verify easily that the capacity and conservation conditions are indeed met and that f is an s-t flow of value k.

Conversely, suppose there is a flow f' in G' of value k. By the integrality theorem for maximum flows (7.14), we know there is an integer-valued flow f of value k; and since all capacities are 1, this means that $f(e)$ is equal to either 0 or 1 for each edge e. Now, consider the set M' of edges of the form (x, y) on which the flow value is 1.

Here are three simple facts about the set M'.

(7.34) *M' contains k edges.*

Proof. To prove this, consider the cut (A, B) in G' with $A = \{s\} \cup X$. The value of the flow is the total flow leaving A, minus the total flow entering A. The first of these terms is simply the cardinality of M', since these are the edges leaving A that carry flow, and each carries exactly one unit of flow. The second of these terms is 0, since there are no edges entering A. Thus, M' contains k edges. ■

(7.35) *Each node in X is the tail of at most one edge in M'.*

Proof. To prove this, suppose $x \in X$ were the tail of at least two edges in M'. Since our flow is integer-valued, this means that at least two units of flow leave from x. By conservation of flow, at least two units of flow would have to come into x—but this is not possible, since only a single edge of capacity 1 enters x. Thus x is the tail of at most one edge in M'. ■

By the same reasoning, we can show

(7.36) *Each node in Y is the head of at most one edge in M'.*

Combining these facts, we see that if we view M' as a set of edges in the original bipartite graph G, we get a matching of size k. In summary, we have proved the following fact.

(7.37) *The size of the maximum matching in G is equal to the value of the maximum flow in G'; and the edges in such a matching in G are the edges that carry flow from X to Y in G'.*

Note the crucial way in which the integrality theorem (7.14) figured in this construction: we needed to know if there is a maximum flow in G' that takes only the values 0 and 1.

Bounding the Running Time Now let's consider how quickly we can compute a maximum matching in G. Let $n = |X| = |Y|$, and let m be the number of edges of G. We'll tacitly assume that there is at least one edge incident to each node in the original problem, and hence $m \geq n/2$. The time to compute a maximum matching is dominated by the time to compute an integer-valued maximum flow in G', since converting this to a matching in G is simple. For this flow problem, we have that $C = \sum_{e \text{ out of } s} c_e = |X| = n$, as s has an edge of capacity 1 to each node of X. Thus, by using the $O(mC)$ bound in (7.5), we get the following.

(7.38) *The Ford-Fulkerson Algorithm can be used to find a maximum matching in a bipartite graph in $O(mn)$ time.*

It's interesting that if we were to use the "better" bounds of $O(m^2 \log_2 C)$ or $O(n^3)$ that we developed in the previous sections, we'd get the inferior running times of $O(m^2 \log n)$ or $O(n^3)$ for this problem. There is nothing contradictory in this. These bounds were designed to be good for *all* instances, even when C is very large relative to m and n. But $C = n$ for the Bipartite Matching Problem, and so the cost of this extra sophistication is not needed.

It is worthwhile to consider what the augmenting paths mean in the network G'. Consider the matching M consisting of edges (x_2, y_2), (x_3, y_3), and (x_5, y_5) in the bipartite graph in Figure 7.1; see also Figure 7.10. Let f be the corresponding flow in G'. This matching is not maximum, so f is not a maximum s-t flow, and hence there is an augmenting path in the residual graph G'_f. One such augmenting path is marked in Figure 7.10(b). Note that the edges (x_2, y_2) and (x_3, y_3) are used backward, and all other edges are used forward. All augmenting paths must alternate between edges used backward and forward, as all edges of the graph G' go from X to Y. Augmenting paths are therefore also called *alternating paths* in the context of finding a maximum matching. The effect of this augmentation is to take the edges used backward out of the matching, and replace them with the edges going forward. Because the augmenting path goes from s to t, there is one more forward edge than backward edge; thus the size of the matching increases by one.

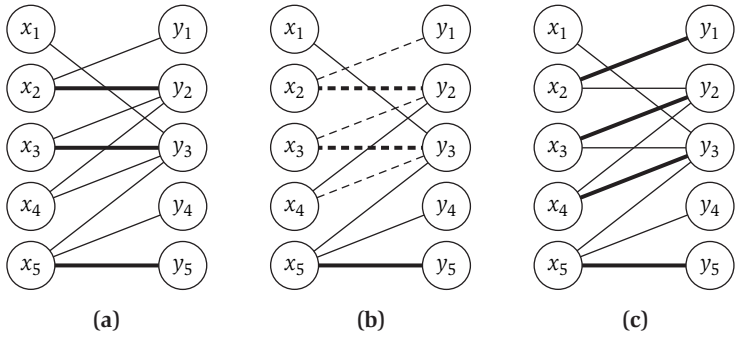

Figure 7.10 (a) A bipartite graph, with a matching M. (b) The augmenting path in the corresponding residual graph. (c) The matching obtained by the augmentation.

Extensions: The Structure of Bipartite Graphs with No Perfect Matching

Algorithmically, we've seen how to find perfect matchings: We use the algorithm above to find a maximum matching and then check to see if this matching is perfect.

But let's ask a slightly less algorithmic question. Not all bipartite graphs have perfect matchings. What does a bipartite graph without a perfect matching look like? Is there an easy way to see that a bipartite graph does not have a perfect matching—or at least an easy way to convince someone the graph has no perfect matching, after we run the algorithm? More concretely, it would be nice if the algorithm, upon concluding that there is no perfect matching, could produce a short "certificate" of this fact. The certificate could allow someone to be quickly convinced that there is no perfect matching, without having to look over a trace of the entire execution of the algorithm.

One way to understand the idea of such a certificate is as follows. We can decide if the graph G has a perfect matching by checking if the maximum flow in a related graph G' has value at least n. By the Max-Flow Min-Cut Theorem, there will be an s-t cut of capacity less than n if the maximum-flow value in G' has value less than n. So, in a way, a cut with capacity less than n provides such a certificate. However, we want a certificate that has a natural meaning in terms of the original graph G.

What might such a certificate look like? For example, if there are nodes $x_1, x_2 \in X$ that have only one incident edge each, and the other end of each edge is the same node y, then clearly the graph has no perfect matching: both x_1 and x_2 would need to get matched to the same node y. More generally, consider a subset of nodes $A \subseteq X$, and let $\Gamma(A) \subseteq Y$ denote the set of all nodes

that are adjacent to nodes in A. If the graph has a perfect matching, then each node in A has to be matched to a different node in $\Gamma(A)$, so $\Gamma(A)$ has to be at least as large as A. This gives us the following fact.

(7.39) *If a bipartite graph $G = (V, E)$ with two sides X and Y has a perfect matching, then for all $A \subseteq X$ we must have $|\Gamma(A)| \geq |A|$.*

This statement suggests a type of certificate demonstrating that a graph does not have a perfect matching: a set $A \subseteq X$ such that $|\Gamma(A)| < |A|$. But is the converse of (7.39) also true? Is it the case that whenever there is no perfect matching, there is a set A like this that proves it? The answer turns out to be yes, provided we add the obvious condition that $|X| = |Y|$ (without which there could certainly not be a perfect matching). This statement is known in the literature as *Hall's Theorem*, though versions of it were discovered independently by a number of different people—perhaps first by König—in the early 1900s. The proof of the statement also provides a way to find such a subset A in polynomial time.

(7.40) *Assume that the bipartite graph $G = (V, E)$ has two sides X and Y such that $|X| = |Y|$. Then the graph G either has a perfect matching or there is a subset $A \subseteq X$ such that $|\Gamma(A)| < |A|$. A perfect matching or an appropriate subset A can be found in $O(mn)$ time.*

Proof. We will use the same graph G' as in (7.37). Assume that $|X| = |Y| = n$. By (7.37) the graph G has a maximum matching if and only if the value of the maximum flow in G' is n.

We need to show that if the value of the maximum flow is less than n, then there is a subset A such that $|\Gamma(A)| < |A|$, as claimed in the statement. By the Max-Flow Min-Cut Theorem (7.12), if the maximum-flow value is less than n, then there is a cut (A', B') with capacity less than n in G'. Now the set A' contains s, and may contain nodes from both X and Y as shown in Figure 7.11. We claim that the set $A = X \cap A'$ has the claimed property. This will prove both parts of the statement, as we've seen in (7.11) that a minimum cut (A', B') can also be found by running the Ford-Fulkerson Algorithm.

First we claim that one can modify the minimum cut (A', B') so as to ensure that $\Gamma(A) \subseteq A'$, where $A = X \cap A'$ as before. To do this, consider a node $y \in \Gamma(A)$ that belongs to B' as shown in Figure 7.11 (a). We claim that by moving y from B' to A', we do not increase the capacity of the cut. For what happens when we move y from B' to A'? The edge (y, t) now crosses the cut, increasing the capacity by one. But previously there was *at least* one edge (x, y) with $x \in A$, since $y \in \Gamma(A)$; all edges from A and y used to cross the cut, and don't anymore. Thus, overall, the capacity of the cut cannot increase. (Note that we

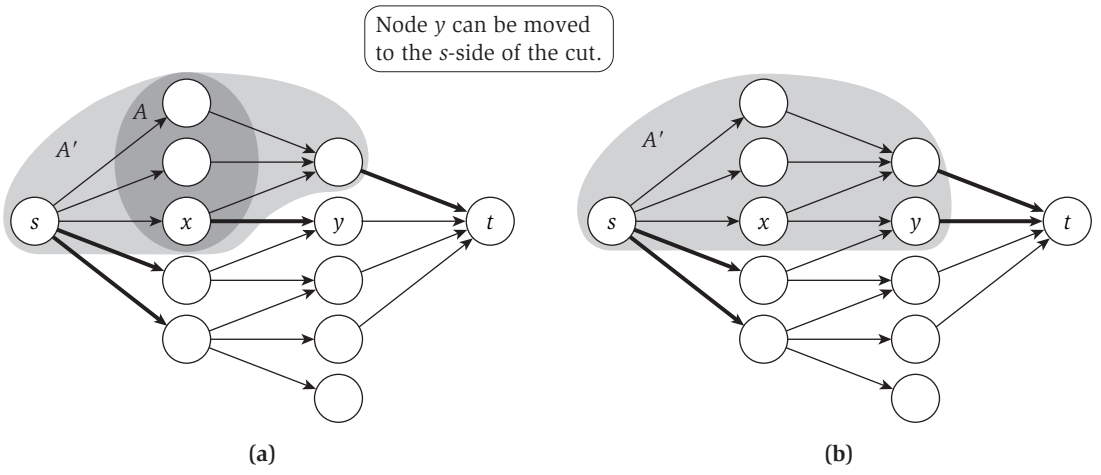

Figure 7.11 (a) A minimum cut in proof of (7.40). (b) The same cut after moving node y to the A' side. The edges crossing the cut are dark.

don't have to be concerned about nodes $x \in X$ that are not in A. The two ends of the edge (x, y) will be on different sides of the cut, but this edge does not add to the capacity of the cut, as it goes from B' to A'.)

Next consider the capacity of this minimum cut (A', B') that has $\Gamma(A) \subseteq A'$ as shown in Figure 7.11(b). Since all neighbors of A belong to A', we see that the only edges out of A' are either edges that leave the source s or that enter the sink t. Thus the capacity of the cut is exactly

$$c(A', B') = |X \cap B'| + |Y \cap A'|.$$

Notice that $|X \cap B'| = n - |A|$, and $|Y \cap A'| \geq |\Gamma(A)|$. Now the assumption that $c(A', B') < n$ implies that

$$n - |A| + |\Gamma(A)| \leq |X \cap B'| + |Y \cap A'| = c(A', B') < n.$$

Comparing the first and the last terms, we get the claimed inequality $|A| > |\Gamma(A)|$. ■

7.6 Disjoint Paths in Directed and Undirected Graphs

In Section 7.1, we described a flow f as a kind of "traffic" in the network. But our actual definition of a flow has a much more static feel to it: For each edge e, we simply specify a number $f(e)$ saying the amount of flow crossing e. Let's see if we can revive the more dynamic, traffic-oriented picture a bit, and try formalizing the sense in which units of flow "travel" from the source to

the sink. From this more dynamic view of flows, we will arrive at something called the *s-t Disjoint Paths Problem*.

The Problem

In defining this problem precisely, we will deal with two issues. First, we will make precise this intuitive correspondence between units of flow traveling along paths, and the notion of flow we've studied so far. Second, we will extend the Disjoint Paths Problem to *undirected* graphs. We'll see that, despite the fact that the Maximum-Flow Problem was defined for a directed graph, it can naturally be used also to handle related problems on undirected graphs.

We say that a set of paths is *edge-disjoint* if their edge sets are disjoint, that is, no two paths share an edge, though multiple paths may go through some of the same nodes. Given a directed graph $G = (V, E)$ with two distinguished nodes $s, t \in V$, the *Directed Edge-Disjoint Paths Problem* is to find the maximum number of edge-disjoint s-t paths in G. The *Undirected Edge-Disjoint Paths Problem* is to find the maximum number of edge-disjoint s-t paths in an undirected graph G. The related question of finding paths that are not only edge-disjoint, but also node-disjoint (of course, other than at nodes s and t) will be considered in the exercises to this chapter.

Designing the Algorithm

Both the directed and the undirected versions of the problem can be solved very naturally using flows. Let's start with the directed problem. Given the graph $G = (V, E)$, with its two distinguished nodes s and t, we define a flow network in which s and t are the source and sink, respectively, and with a capacity of 1 on each edge. Now suppose there are k edge-disjoint s-t paths. We can make each of these paths carry one unit of flow: We set the flow to be $f(e) = 1$ for each edge e on any of the paths, and $f(e') = 0$ on all other edges, and this defines a feasible flow of value k.

(7.41) *If there are k edge-disjoint paths in a directed graph G from s to t, then the value of the maximum s-t flow in G is at least k.*

Suppose we could show the converse to (7.41) as well: If there is a flow of value k, then there exist k edge-disjoint s-t paths. Then we could simply compute a maximum s-t flow in G and declare (correctly) this to be the maximum number of edge-disjoint s-t paths.

We now proceed to prove this converse statement, confirming that this approach using flow indeed gives us the correct answer. Our analysis will also provide a way to extract k edge-disjoint paths from an integer-valued flow sending k units from s to t. Thus computing a maximum flow in G will

not only give us the maximum *number* of edge-disjoint paths, but the paths as well.

Analyzing the Algorithm

Proving the converse direction of (7.41) is the heart of the analysis, since it will immediately establish the optimality of the flow-based algorithm to find disjoint paths.

To prove this, we will consider a flow of value at least k, and construct k edge-disjoint paths. By (7.14), we know that there is a maximum flow f with integer flow values. Since all edges have a capacity bound of 1, and the flow is integer-valued, each edge that carries flow under f has exactly one unit of flow on it. Thus we just need to show the following.

(7.42) *If f is a 0-1 valued flow of value v, then the set of edges with flow value $f(e) = 1$ contains a set of v edge-disjoint paths.*

Proof. We prove this by induction on the number of edges in f that carry flow. If $v = 0$, there is nothing to prove. Otherwise, there must be an edge (s, u) that carries one unit of flow. We now "trace out" a path of edges that must also carry flow: Since (s, u) carries a unit of flow, it follows by conservation that there is some edge (u, v) that carries one unit of flow, and then there must be an edge (v, w) that carries one unit of flow, and so forth. If we continue in this way, one of two things will eventually happen: Either we will reach t, or we will reach a node v for the second time.

If the first case happens—we find a path P from s to t—then we'll use this path as one of our v paths. Let f' be the flow obtained by decreasing the flow values on the edges along P to 0. This new flow f' has value $v - 1$, and it has fewer edges that carry flow. Applying the induction hypothesis for f', we get $v - 1$ edge-disjoint paths, which, along with path P, form the v paths claimed.

If P reaches a node v for the second time, then we have a situation like the one pictured in Figure 7.12. (The edges in the figure all carry one unit of flow, and the dashed edges indicate the path traversed so far, which has just reached a node v for the second time.) In this case, we can make progress in a different way.

Consider the cycle C of edges visited between the first and second appearances of v. We obtain a new flow f' from f by decreasing the flow values on the edges along C to 0. This new flow f' has value v, but it has fewer edges that carry flow. Applying the induction hypothesis for f', we get the v edge-disjoint paths as claimed. ■

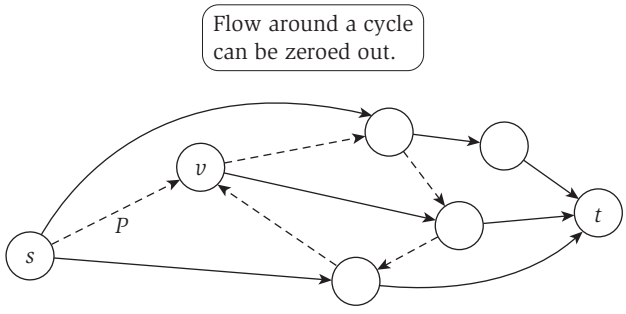

Figure 7.12 The edges in the figure all carry one unit of flow. The path P of dashed edges is one possible path in the proof of (7.42).

We can summarize (7.41) and (7.42) in the following result.

(7.43) *There are k edge-disjoint paths in a directed graph G from s to t if and only if the value of the maximum value of an s-t flow in G is at least k.*

Notice also how the proof of (7.42) provides an actual procedure for constructing the k paths, given an integer-valued maximum flow in G. This procedure is sometimes referred to as a *path decomposition* of the flow, since it "decomposes" the flow into a constituent set of paths. Hence we have shown that our flow-based algorithm finds the maximum number of edge-disjoint s-t paths and also gives us a way to construct the actual paths.

Bounding the Running Time For this flow problem, $C = \sum_{e \text{ out of } s} c_e \leq |V| = n$, as there are at most $|V|$ edges out of s, each of which has capacity 1. Thus, by using the $O(mC)$ bound in (7.5), we get an integer maximum flow in $O(mn)$ time.

The path decomposition procedure in the proof of (7.42), which produces the paths themselves, can also be made to run in $O(mn)$ time. To see this, note that this procedure, with a little care, can produce a single path from s to t using at most constant work per edge in the graph, and hence in $O(m)$ time. Since there can be at most $n - 1$ edge-disjoint paths from s to t (each must use a different edge out of s), it therefore takes time $O(mn)$ to produce all the paths.

In summary, we have shown

(7.44) *The Ford-Fulkerson Algorithm can be used to find a maximum set of edge-disjoint s-t paths in a directed graph G in $O(mn)$ time.*

A Version of the Max-Flow Min-Cut Theorem for Disjoint Paths The Max-Flow Min-Cut Theorem (7.13) can be used to give the following characteri-

zation of the maximum number of edge-disjoint s-t paths. We say that a set $F \subseteq E$ of edges *separates* s from t if, after removing the edges F from the graph G, no s-t paths remain in the graph.

(7.45) *In every directed graph with nodes s and t, the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal separates s from t.*

Proof. If the removal of a set $F \subseteq E$ of edges separates s from t, then each s-t path must use at least one edge from F, and hence the number of edge-disjoint s-t paths is at most $|F|$.

To prove the other direction, we will use the Max-Flow Min-Cut Theorem (7.13). By (7.43) the maximum number of edge-disjoint paths is the value v of the maximum s-t flow. Now (7.13) states that there is an s-t cut (A, B) with capacity v. Let F be the set of edges that go from A to B. Each edge has capacity 1, so $|F| = v$ and, by the definition of an s-t cut, removing these v edges from G separates s from t. ■

This result, then, can be viewed as the natural special case of the Max-Flow Min-Cut Theorem in which all edge capacities are equal to 1. In fact, this special case was proved by Menger in 1927, much before the full Max-Flow Min-Cut Theorem was formulated and proved; for this reason, (7.45) is often called *Menger's Theorem*. If we think about it, the proof of Hall's Theorem (7.40) for bipartite matchings involves a reduction to a graph with unit-capacity edges, and so it can be proved using Menger's Theorem rather than the general Max-Flow Min-Cut Theorem. In other words, Hall's Theorem is really a special case of Menger's Theorem, which in turn is a special case of the Max-Flow Min-Cut Theorem. And the history follows this progression, since they were discovered in this order, a few decades apart.[2]

Extensions: Disjoint Paths in Undirected Graphs

Finally, we consider the disjoint paths problem in an undirected graph G. Despite the fact that our graph G is now undirected, we can use the maximum-flow algorithm to obtain edge-disjoint paths in G. The idea is quite simple: We replace each undirected edge (u, v) in G by two directed edges (u, v) and

[2] In fact, in an interesting retrospective written in 1981, Menger relates his version of the story of how he first explained his theorem to König, one of the independent discoverers of Hall's Theorem. You might think that König, having thought a lot about these problems, would have immediately grasped why Menger's generalization of his theorem was true, and perhaps even considered it obvious. But, in fact, the opposite happened; König didn't believe it could be right and stayed up all night searching for a counterexample. The next day, exhausted, he sought out Menger and asked him for the proof.

(v, u), and in this way create a directed version G' of G. (We may delete the edges into s and out of t, since they are not useful.) Now we want to use the Ford-Fulkerson Algorithm in the resulting directed graph. However, there is an important issue we need to deal with first. Notice that two paths P_1 and P_2 may be edge-disjoint in the directed graph and yet share an edge in the undirected graph G: This happens if P_1 uses directed edge (u, v) while P_2 uses edge (v, u). However, it is not hard to see that there always exists a maximum flow in any network that uses at most *one* out of each pair of oppositely directed edges.

(7.46) *In any flow network, there is a maximum flow f where for all opposite directed edges $e = (u, v)$ and $e' = (v, u)$, either $f(e) = 0$ or $f(e') = 0$. If the capacities of the flow network are integral, then there also is such an integral maximum flow.*

Proof. We consider any maximum flow f, and we modify it to satisfy the claimed condition. Assume $e = (u, v)$ and $e' = (v, u)$ are opposite directed edges, and $f(e) \neq 0$, $f(e') \neq 0$. Let δ be the smaller of these values, and modify f by decreasing the flow value on both e and e' by δ. The resulting flow f' is feasible, has the same value as f, and its value on one of e and e' is 0. ■

Now we can use the Ford-Fulkerson Algorithm and the path decomposition procedure from (7.42) to obtain edge-disjoint paths in the undirected graph G.

(7.47) *There are k edge-disjoint paths in an undirected graph G from s to t if and only if the maximum value of an s-t flow in the directed version G' of G is at least k. Furthermore, the Ford-Fulkerson Algorithm can be used to find a maximum set of disjoint s-t paths in an undirected graph G in $O(mn)$ time.*

The undirected analogue of (7.45) is also true, as in any s-t cut, at most one of the two oppositely directed edges can cross from the s-side to the t-side of the cut (for if one crosses, then the other must go from the t-side to the s-side).

(7.48) *In every undirected graph with nodes s and t, the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal separates s from t.*

7.7 Extensions to the Maximum-Flow Problem

Much of the power of the Maximum-Flow Problem has essentially nothing to do with the fact that it models traffic in a network. Rather, it lies in the fact that many problems with a nontrivial combinatorial search component can

be solved in polynomial time because they can be reduced to the problem of finding a maximum flow or a minimum cut in a directed graph.

Bipartite Matching is a natural first application in this vein; in the coming sections, we investigate a range of further applications. To begin with, we stay with the picture of flow as an abstract kind of "traffic," and look for more general conditions we might impose on this traffic. These more general conditions will turn out to be useful for some of our further applications.

In particular, we focus on two generalizations of maximum flow. We will see that both can be reduced to the basic Maximum-Flow Problem.

The Problem: Circulations with Demands

One simplifying aspect of our initial formulation of the Maximum-Flow Problem is that we had only a single source s and a single sink t. Now suppose that there can be a set S of sources generating flow, and a set T of sinks that can absorb flow. As before, there is an integer capacity on each edge.

With multiple sources and sinks, it is a bit unclear how to decide which source or sink to favor in a maximization problem. So instead of maximizing the flow value, we will consider a problem where sources have fixed *supply* values and sinks have fixed *demand* values, and our goal is to ship flow from nodes with available supply to those with given demands. Imagine, for example, that the network represents a system of highways or railway lines in which we want to ship products from factories (which have supply) to retail outlets (which have demand). In this type of problem, we will not be seeking to maximize a particular value; rather, we simply want to satisfy all the demand using the available supply.

Thus we are given a flow network $G = (V, E)$ with capacities on the edges. Now, associated with each node $v \in V$ is a *demand* d_v. If $d_v > 0$, this indicates that the node v has a *demand* of d_v for flow; the node is a sink, and it wishes to receive d_v units more flow than it sends out. If $d_v < 0$, this indicates that v has a *supply* of $-d_v$; the node is a source, and it wishes to send out $-d_v$ units more flow than it receives. If $d_v = 0$, then the node v is neither a source nor a sink. We will assume that all capacities and demands are integers.

We use S to denote the set of all nodes with negative demand and T to denote the set of all nodes with positive demand. Although a node v in S wants to send out more flow than it receives, it will be okay for it to have flow that enters on incoming edges; it should just be more than compensated by the flow that leaves v on outgoing edges. The same applies (in the opposite direction) to the set T.

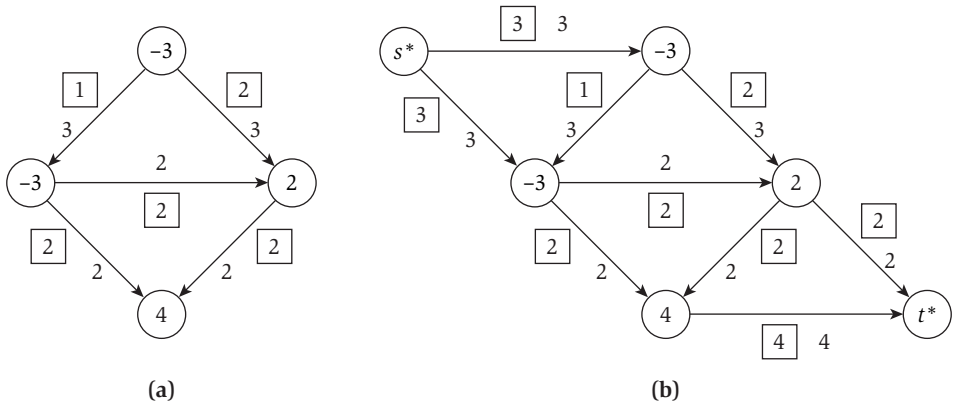

Figure 7.13 (a) An instance of the Circulation Problem together with a solution: Numbers inside the nodes are demands; numbers labeling the edges are capacities and flow values, with the flow values inside boxes. (b) The result of reducing this instance to an equivalent instance of the Maximum-Flow Problem.

In this setting, we say that a *circulation* with demands $\{d_v\}$ is a function f that assigns a nonnegative real number to each edge and satisfies the following two conditions.

(i) *(Capacity conditions)* For each $e \in E$, we have $0 \leq f(e) \leq c_e$.

(ii) *(Demand conditions)* For each $v \in V$, we have v, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

Now, instead of considering a maximization problem, we are concerned with a *feasibility problem*: We want to know whether there *exists* a circulation that meets conditions (i) and (ii).

For example, consider the instance in Figure 7.13(a). Two of the nodes are sources, with demands -3 and -3; and two of the nodes are sinks, with demands 2 and 4. The flow values in the figure constitute a feasible circulation, indicating how all demands can be satisfied while respecting the capacities.

If we consider an arbitrary instance of the Circulation Problem, here is a simple condition that must hold in order for a feasible circulation to exist: The total supply must equal the total demand.

(7.49) *If there exists a feasible circulation with demands $\{d_v\}$, then $\sum_v d_v = 0$.*

Proof. Suppose there exists a feasible circulation f in this setting. Then $\sum_v d_v = \sum_v f^{\text{in}}(v) - f^{\text{out}}(v)$. Now, in this latter expression, the value $f(e)$ for each edge $e = (u, v)$ is counted exactly twice: once in $f^{\text{out}}(u)$ and once in $f^{\text{in}}(v)$. These two terms cancel out; and since this holds for all values $f(e)$, the overall sum is 0. ■

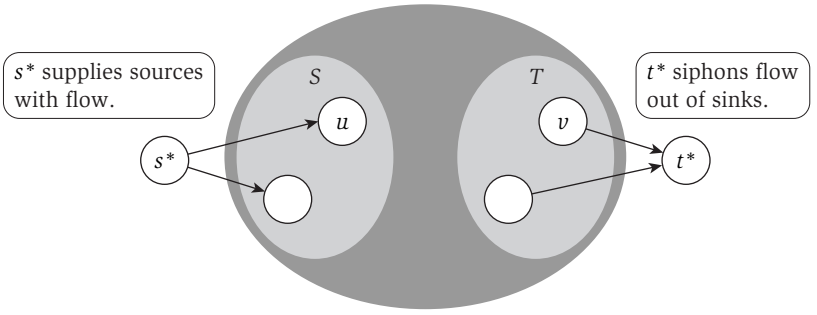

Figure 7.14 Reducing the Circulation Problem to the Maximum-Flow Problem.

Thanks to (7.49), we know that

$$\sum_{v:d_v>0} d_v = \sum_{v:d_v<0} -d_v.$$

Let D denote this common value.

Designing and Analyzing an Algorithm for Circulations

It turns out that we can reduce the problem of finding a feasible circulation with demands $\{d_v\}$ to the problem of finding a maximum s-t flow in a different network, as shown in Figure 7.14.

The reduction looks very much like the one we used for Bipartite Matching: we attach a "super-source" s^* to each node in S, and a "super-sink" t^* to each node in T. More specifically, we create a graph G' from G by adding new nodes s^* and t^* to G. For each node $v \in T$—that is, each node v with $d_v > 0$—we add an edge (v, t^*) with capacity d_v. For each node $u \in S$—that is, each node with $d_u < 0$—we add an edge (s^*, u) with capacity $-d_u$. We carry the remaining structure of G over to G' unchanged.

In this graph G', we will be seeking a maximum s^*-t^* flow. Intuitively, we can think of this reduction as introducing a node s^* that "supplies" all the sources with their extra flow, and a node t^* that "siphons" the extra flow out of the sinks. For example, part (b) of Figure 7.13 shows the result of applying this reduction to the instance in part (a).

Note that there cannot be an s^*-t^* flow in G' of value greater than D, since the cut (A, B) with $A = \{s^*\}$ only has capacity D. Now, if there is a feasible circulation f with demands $\{d_v\}$ in G, then by sending a flow value of $-d_v$ on each edge (s^*, v), and a flow value of d_v on each edge (v, t^*), we obtain an s^*-t^* flow in G' of value D, and so this is a maximum flow. Conversely, suppose there is a (maximum) s^*-t^* flow in G' of value D. It must be that every edge

out of s^*, and every edge into t^*, is completely saturated with flow. Thus, if we delete these edges, we obtain a circulation f in G with $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ for each node v. Further, if there is a flow of value D in G', then there is such a flow that takes integer values.

In summary, we have proved the following.

(7.50) *There is a feasible circulation with demands $\{d_v\}$ in G if and only if the maximum s^*-t^* flow in G' has value D. If all capacities and demands in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer-valued.*

At the end of Section 7.5, we used the Max-Flow Min-Cut Theorem to derive the characterization (7.40) of bipartite graphs that do not have perfect matchings. We can give an analogous characterization for graphs that do not have a feasible circulation. The characterization uses the notion of a *cut*, adapted to the present setting. In the context of circulation problems with demands, a cut (A, B) is any partition of the node set V into two sets, with no restriction on which side of the partition the sources and sinks fall. We include the characterization here without a proof.

(7.51) *The graph G has a feasible circulation with demands $\{d_v\}$ if and only if for all cuts (A, B),*

$$\sum_{v \in B} d_v \leq c(A, B).$$

It is important to note that our network has only a single "kind" of flow. Although the flow is supplied from multiple sources, and absorbed at multiple sinks, we cannot place restrictions on which source will supply the flow to which sink; we have to let our algorithm decide this. A harder problem is the *Multicommodity Flow Problem*; here sink t_i must be supplied with flow that originated at source s_i, for each i. We will discuss this issue further in Chapter 11.

The Problem: Circulations with Demands and Lower Bounds

Finally, let us generalize the previous problem a little. In many applications, we not only want to satisfy demands at various nodes; we also want to force the flow to make use of certain edges. This can be enforced by placing *lower bounds* on edges, as well as the usual upper bounds imposed by edge capacities.

Consider a flow network $G = (V, E)$ with a *capacity* c_e and a *lower bound* ℓ_e on each edge e. We will assume $0 \leq \ell_e \leq c_e$ for each e. As before, each node v will also have a demand d_v, which can be either positive or negative. We will assume that all demands, capacities, and lower bounds are integers.

The given quantities have the same meaning as before, and now a lower bound ℓ_e means that the flow value on e must be *at least* ℓ_e. Thus a circulation in our flow network must satisfy the following two conditions.

(i) *(Capacity conditions)* For each $e \in E$, we have $\ell_e \leq f(e) \leq c_e$.

(ii) *(Demand conditions)* For every $v \in V$, we have $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

As before, we wish to decide whether there exists a *feasible circulation*—one that satisfies these conditions.

Designing and Analyzing an Algorithm with Lower Bounds

Our strategy will be to reduce this to the problem of finding a circulation with demands but no lower bounds. (We've seen that this latter problem, in turn, can be reduced to the standard Maximum-Flow Problem.) The idea is as follows. We know that on each edge e, we need to send at least ℓ_e units of flow. So suppose that we define an initial circulation f_0 simply by $f_0(e) = \ell_e$. f_0 satisfies all the capacity conditions (both lower and upper bounds); but it presumably does not satisfy all the demand conditions. In particular,

$$f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{e \text{ into } v} \ell_e - \sum_{e \text{ out of } v} \ell_e.$$

Let us denote this quantity by L_v. If $L_v = d_v$, then we have satisfied the demand condition at v; but if not, then we need to superimpose a circulation f_1 on top of f_0 that will clear the remaining "imbalance" at v. So we need $f_1^{\text{in}}(v) - f_1^{\text{out}}(v) = d_v - L_v$. And how much capacity do we have with which to do this? Having already sent ℓ_e units of flow on each edge e, we have $c_e - \ell_e$ more units to work with.

These considerations directly motivate the following construction. Let the graph G' have the same nodes and edges, with capacities and demands, but no lower bounds. The capacity of edge e will be $c_e - \ell_e$. The demand of node v will be $d_v - L_v$.

For example, consider the instance in Figure 7.15(a). This is the same as the instance we saw in Figure 7.13, except that we have now given one of the edges a lower bound of 2. In part (b) of the figure, we eliminate this lower bound by sending two units of flow across the edge. This reduces the upper bound on the edge and changes the demands at the two ends of the edge. In the process, it becomes clear that there is no feasible circulation, since after applying the construction there is a node with a demand of -5, and a total of only four units of capacity on its outgoing edges.

We now claim that our general construction produces an equivalent instance with demands but no lower bounds; we can therefore use our algorithm for this latter problem.

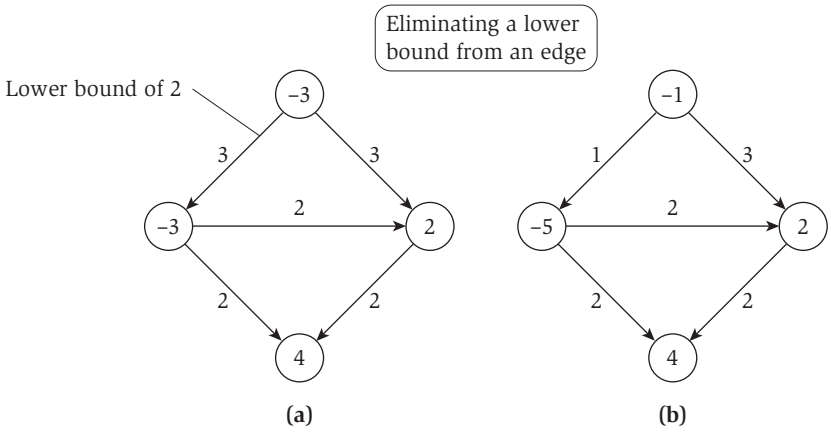

Figure 7.15 (a) An instance of the Circulation Problem with lower bounds: Numbers inside the nodes are demands, and numbers labeling the edges are capacities. We also assign a lower bound of 2 to one of the edges. (b) The result of reducing this instance to an equivalent instance of the Circulation Problem without lower bounds.

(7.52) *There is a feasible circulation in G if and only if there is a feasible circulation in G'. If all demands, capacities, and lower bounds in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer-valued.*

Proof. First suppose there is a circulation f' in G'. Define a circulation f in G by $f(e) = f'(e) + \ell_e$. Then f satisfies the capacity conditions in G, and

$$f^{\text{in}}(v) - f^{\text{out}}(v) = \sum_{e \text{ into } v} (\ell_e + f'(e)) - \sum_{e \text{ out of } v} (\ell_e + f'(e)) = L_v + (d_v - L_v) = d_v,$$

so it satisfies the demand conditions in G as well.

Conversely, suppose there is a circulation f in G, and define a circulation f' in G' by $f'(e) = f(e) - \ell_e$. Then f' satisfies the capacity conditions in G', and

$$(f')^{\text{in}}(v) - (f')^{\text{out}}(v) = \sum_{e \text{ into } v} (f(e) - \ell_e) - \sum_{e \text{ out of } v} (f(e) - \ell_e) = d_v - L_v,$$

so it satisfies the demand conditions in G' as well. ■

7.8 Survey Design

Many problems that arise in applications can, in fact, be solved efficiently by a reduction to Maximum Flow, but it is often difficult to discover when such a reduction is possible. In the next few sections, we give several paradigmatic examples of such problems. The goal is to indicate what such reductions tend

to look like and to illustrate some of the most common uses of flows and cuts in the design of efficient combinatorial algorithms. One point that will emerge is the following: Sometimes the solution one wants involves the computation of a maximum flow, and sometimes it involves the computation of a minimum cut; both flows and cuts are very useful algorithmic tools.

We begin with a basic application that we call *survey design*, a simple version of a task faced by many companies wanting to measure customer satisfaction. More generally, the problem illustrates how the construction used to solve the Bipartite Matching Problem arises naturally in any setting where we want to carefully balance decisions across a set of options—in this case, designing questionnaires by balancing relevant questions across a population of consumers.

The Problem

A major issue in the burgeoning field of *data mining* is the study of consumer preference patterns. Consider a company that sells k products and has a database containing the purchase histories of a large number of customers. (Those of you with "Shopper's Club" cards may be able to guess how this data gets collected.) The company wishes to conduct a survey, sending customized questionnaires to a particular group of n of its customers, to try determining which products people like overall.

Here are the guidelines for designing the survey.

- Each customer will receive questions about a certain subset of the products.
- A customer can only be asked about products that he or she has purchased.
- To make each questionnaire informative, but not too long so as to discourage participation, each customer i should be asked about a number of products between c_i and c_i'.
- Finally, to collect sufficient data about each product, there must be between p_j and p_j' distinct customers asked about each product j.

More formally, the input to the *Survey Design Problem* consists of a bipartite graph G whose nodes are the customers and the products, and there is an edge between customer i and product j if he or she has ever purchased product j. Further, for each customer $i = 1, \ldots, n$, we have limits $c_i \leq c_i'$ on the number of products he or she can be asked about; for each product $j = 1, \ldots, k$, we have limits $p_j \leq p_j'$ on the number of distinct customers that have to be asked about it. The problem is to decide if there is a way to design a questionnaire for each customer so as to satisfy all these conditions.

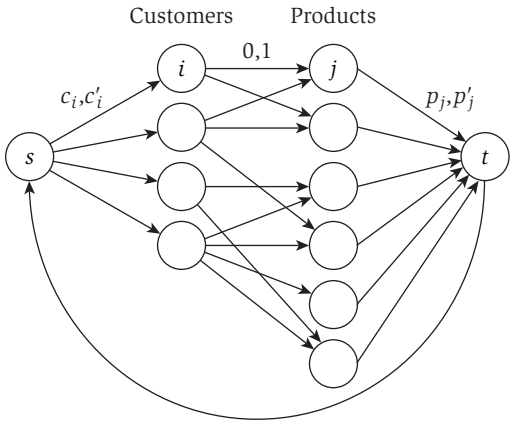

Figure 7.16 The Survey Design Problem can be reduced to the problem of finding a feasible circulation: Flow passes from customers (with capacity bounds indicating how many questions they can be asked) to products (with capacity bounds indicating how many questions should be asked about each product).

Designing the Algorithm

We will solve this problem by reducing it to a circulation problem on a flow network G' with demands and lower bounds as shown in Figure 7.16. To obtain the graph G' from G, we orient the edges of G from customers to products, add nodes s and t with edges (s, i) for each customer $i = 1, \ldots, n$, edges (j, t) for each product $j = 1, \ldots, k$, and an edge (t, s). The circulation in this network will correspond to the way in which questions are asked. The flow on the edge (s, i) is the number of products included on the questionnaire for customer i, so this edge will have a capacity of c'_i and a lower bound of c_i. The flow on the edge (j, t) will correspond to the number of customers who were asked about product j, so this edge will have a capacity of p'_j and a lower bound of p_j. Each edge (i, j) going from a customer to a product he or she bought has capacity 1, and 0 as the lower bound. The flow carried by the edge (t, s) corresponds to the overall number of questions asked. We can give this edge a capacity of $\sum_i c'_i$ and a lower bound of $\sum_i c_i$. All nodes have demand 0.

Our algorithm is simply to construct this network G' and check whether it has a feasible circulation. We now formulate a claim that establishes the correctness of this algorithm.

Analyzing the Algorithm

(7.53) *The graph G' just constructed has a feasible circulation if and only if there is a feasible way to design the survey.*

Proof. The construction above immediately suggests a way to turn a survey design into the corresponding flow. The edge (i, j) will carry one unit of flow if customer i is asked about product j in the survey, and will carry no flow otherwise. The flow on the edges (s, i) is the number of questions asked from customer i, the flow on the edge (j, t) is the number of customers who were asked about product j, and finally, the flow on edge (t, s) is the overall number of questions asked. This flow satisfies the 0 demand, that is, there is flow conservation at every node. If the survey satisfies these rules, then the corresponding flow satisfies the capacities and lower bounds.

Conversely, if the Circulation Problem is feasible, then by (7.52) there is a feasible circulation that is integer-valued, and such an integer-valued circulation naturally corresponds to a feasible survey design. Customer i will be surveyed about product j if and only if the edge (i, j) carries a unit of flow. ■

7.9 Airline Scheduling

The computational problems faced by the nation's large airline carriers are almost too complex to even imagine. They have to produce schedules for thousands of routes each day that are efficient in terms of equipment usage, crew allocation, customer satisfaction, and a host of other factors—all in the face of unpredictable issues like weather and breakdowns. It's not surprising that they're among the largest consumers of high-powered algorithmic techniques.

Covering these computational problems in any realistic level of detail would take us much too far afield. Instead, we'll discuss a "toy" problem that captures, in a very clean way, some of the resource allocation issues that arise in a context such as this. And, as is common in this book, the toy problem will be much more useful for our purposes than the "real" problem, for the solution to the toy problem involves a very general technique that can be applied in a wide range of situations.

The Problem

Suppose you're in charge of managing a fleet of airplanes and you'd like to create a flight schedule for them. Here's a very simple model for this. Your market research has identified a set of m particular flight segments that would be very lucrative if you could serve them; flight segment j is specified by four parameters: its origin airport, its destination airport, its departure time, and its arrival time. Figure 7.17(a) shows a simple example, consisting of six flight segments you'd like to serve with your planes over the course of a single day:

(1) Boston (depart 6 A.M.) – Washington DC (arrive 7 A.M.)
(2) Philadelphia (depart 7 A.M.) – Pittsburgh (arrive 8 A.M.)

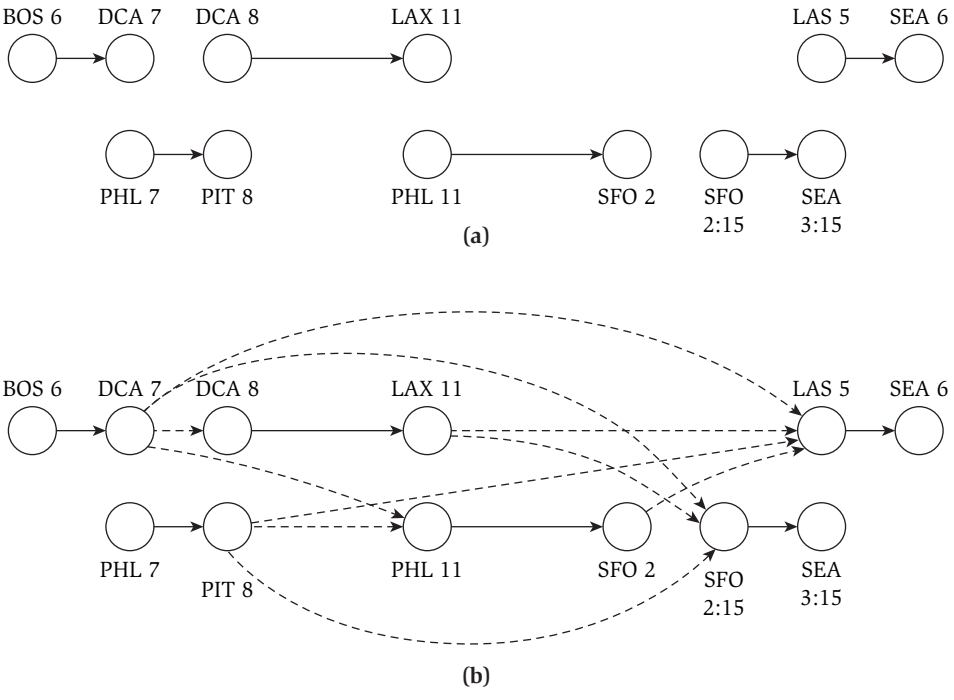

Figure 7.17 (a) A small instance of our simple Airline Scheduling Problem. (b) An expanded graph showing which flights are reachable from which others.

(3) Washington DC (depart 8 A.M.) – Los Angeles (arrive 11 A.M.)
(4) Philadelphia (depart 11 A.M.) – San Francisco (arrive 2 P.M.)
(5) San Francisco (depart 2:15 P.M.) – Seattle (arrive 3:15 P.M.)
(6) Las Vegas (depart 5 P.M.) – Seattle (arrive 6 P.M.)

Note that each segment includes the times you want the flight to serve as well as the airports.

It is possible to use a single plane for a flight segment i, and then later for a flight segment j, provided that

(a) the destination of i is the same as the origin of j, and there's enough time to perform maintenance on the plane in between; or

(b) you can add a flight segment in between that gets the plane from the destination of i to the origin of j with adequate time in between.

For example, assuming an hour for intermediate maintenance time, you could use a single plane for flights (1), (3), and (6) by having the plane sit in Washington, DC, between flights (1) and (3), and then inserting the flight

Los Angeles (depart 12 noon) – Las Vegas (1 P.M.)

in between flights (3) and (6).

Formulating the Problem We can model this situation in a very general way as follows, abstracting away from specific rules about maintenance times and intermediate flight segments: We will simply say that flight j is *reachable* from flight i if it is possible to use the same plane for flight i, and then later for flight j as well. So under our specific rules (a) and (b) above, we can easily determine for each pair i, j whether flight j is reachable from flight i. (Of course, one can easily imagine more complex rules for reachability. For example, the length of maintenance time needed in (a) might depend on the airport; or in (b) we might require that the flight segment you insert be sufficiently profitable on its own.) But the point is that we can handle any set of rules with our definition: The input to the problem will include not just the flight segments, but also a specification of the pairs (i, j) for which a later flight j is reachable from an earlier flight i. These pairs can form an arbitrary directed acyclic graph.

The goal in this problem is to determine whether it's possible to serve all m flights on your original list, using at most k planes total. In order to do this, you need to find a way of efficiently reusing planes for multiple flights.

For example, let's go back to the instance in Figure 7.17 and assume we have $k = 2$ planes. If we use one of the planes for flights (1), (3), and (6) as proposed above, we wouldn't be able to serve all of flights (2), (4), and (5) with the other (since there wouldn't be enough maintenance time in San Francisco between flights (4) and (5)). However, there is a way to serve all six flights using two planes, via a different solution: One plane serves flights (1), (3), and (5) (splicing in an LAX–SFO flight), while the other serves (2), (4), and (6) (splicing in PIT–PHL and SFO–LAS).

Designing the Algorithm

We now discuss an efficient algorithm that can solve arbitrary instances of the Airline Scheduling Problem, based on network flow. We will see that flow techniques adapt very naturally to this problem.

The solution is based on the following idea. Units of flow will correspond to airplanes. We will have an edge for each flight, and upper and lower capacity bounds of 1 on these edges to require that exactly one unit of flow crosses this edge. In other words, each flight must be served by one of the planes. If (u_i, v_i) is the edge representing flight i, and (u_j, v_j) is the edge representing flight j, and flight j is reachable from flight i, then we will have an edge from v_i to u_j

with capacity 1; in this way, a unit of flow can traverse (u_i, v_i) and then move directly to (u_j, v_j). Such a construction of edges is shown in Figure 7.17(b).

We extend this to a flow network by including a source and sink; we now give the full construction in detail. The node set of the underlying graph G is defined as follows.

- For each flight i, the graph G will have the two nodes u_i and v_i.
- G will also have a distinct source node s and sink node t.

The edge set of G is defined as follows.

- For each i, there is an edge (u_i, v_i) with a lower bound of 1 and a capacity of 1. *(Each flight on the list must be served.)*
- For each i and j so that flight j is reachable from flight i, there is an edge (v_i, u_j) with a lower bound of 0 and a capacity of 1. *(The same plane can perform flights i and j.)*
- For each i, there is an edge (s, u_i) with a lower bound of 0 and a capacity of 1. *(Any plane can begin the day with flight i.)*
- For each j, there is an edge (v_j, t) with a lower bound of 0 and a capacity of 1. *(Any plane can end the day with flight j.)*
- There is an edge (s, t) with lower bound 0 and capacity k. *(If we have extra planes, we don't need to use them for any of the flights.)*

Finally, the node s will have a demand of $-k$, and the node t will have a demand of k. All other nodes will have a demand of 0.

Our algorithm is to construct the network G and search for a feasible circulation in it. We now prove the correctness of this algorithm.

Analyzing the Algorithm

(7.54) *There is a way to perform all flights using at most k planes if and only if there is a feasible circulation in the network G.*

Proof. First, suppose there is a way to perform all flights using $k' \leq k$ planes. The set of flights performed by each individual plane defines a path P in the network G, and we send one unit of flow on each such path P. To satisfy the full demands at s and t, we send $k - k'$ units of flow on the edge (s, t). The resulting circulation satisfies all demand, capacity, and lower bound conditions.

Conversely, consider a feasible circulation in the network G. By (7.52), we know that there is a feasible circulation with integer flow values. Suppose that k' units of flow are sent on edges other than (s, t). Since all other edges have a capacity bound of 1, and the circulation is integer-valued, each such edge that carries flow has exactly one unit of flow on it.

We now convert this to a schedule using the same kind of construction we saw in the proof of (7.42), where we converted a flow to a collection of paths. In fact, the situation is easier here since the graph has no cycles. Consider an edge (s, u_i) that carries one unit of flow. It follows by conservation that (u_i, v_i) carries one unit of flow, and that there is a unique edge out of v_i that carries one unit of flow. If we continue in this way, we construct a path P from s to t, so that each edge on this path carries one unit of flow. We can apply this construction to each edge of the form (s, u_j) carrying one unit of flow; in this way, we produce k' paths from s to t, each consisting of edges that carry one unit of flow. Now, for each path P we create in this way, we can assign a single plane to perform all the flights contained in this path. ■

Extensions: Modeling Other Aspects of the Problem

Airline scheduling consumes countless hours of CPU time in real life. We mentioned at the beginning, however, that our formulation here is really a toy problem; it ignores several obvious factors that would have to be taken into account in these applications. First of all, it ignores the fact that a given plane can only fly a certain number of hours before it needs to be temporarily taken out of service for more significant maintenance. Second, we are making up an optimal schedule for a single day (or at least for a single span of time) as though there were no yesterday or tomorrow; in fact we also need the planes to be optimally positioned for the start of day $N + 1$ at the end of day N. Third, all these planes need to be staffed by flight crews, and while crews are also reused across multiple flights, a whole different set of constraints operates here, since human beings and airplanes experience fatigue at different rates. And these issues don't even begin to cover the fact that serving any particular flight segment is not a hard constraint; rather, the real goal is to optimize revenue, and so we can pick and choose among many possible flights to include in our schedule (not to mention designing a good fare structure for passengers) in order to achieve this goal.

Ultimately, the message is probably this: Flow techniques are useful for solving problems of this type, and they are genuinely used in practice. Indeed, our solution above is a general approach to the efficient reuse of a limited set of resources in many settings. At the same time, running an airline efficiently in real life is a very difficult problem.

7.10 Image Segmentation

A central problem in image processing is the *segmentation* of an image into various coherent regions. For example, you may have an image representing a picture of three people standing in front of a complex background scene. A

natural but difficult goal is to identify each of the three people as coherent objects in the scene.

The Problem

One of the most basic problems to be considered along these lines is that of foreground/background segmentation: We wish to label each pixel in an image as belonging to either the foreground of the scene or the background. It turns out that a very natural model here leads to a problem that can be solved efficiently by a minimum cut computation.

Let V be the set of *pixels* in the underlying image that we're analyzing. We will declare certain pairs of pixels to be *neighbors*, and use E to denote the set of all pairs of neighboring pixels. In this way, we obtain an *undirected* graph $G = (V, E)$. We will be deliberately vague on what exactly we mean by a "pixel," or what we mean by the "neighbor" relation. In fact, any graph G will yield an efficiently solvable problem, so we are free to define these notions in any way that we want. Of course, it is natural to picture the pixels as constituting a grid of dots, and the neighbors of a pixel to be those that are directly adjacent to it in this grid, as shown in Figure 7.18(a).

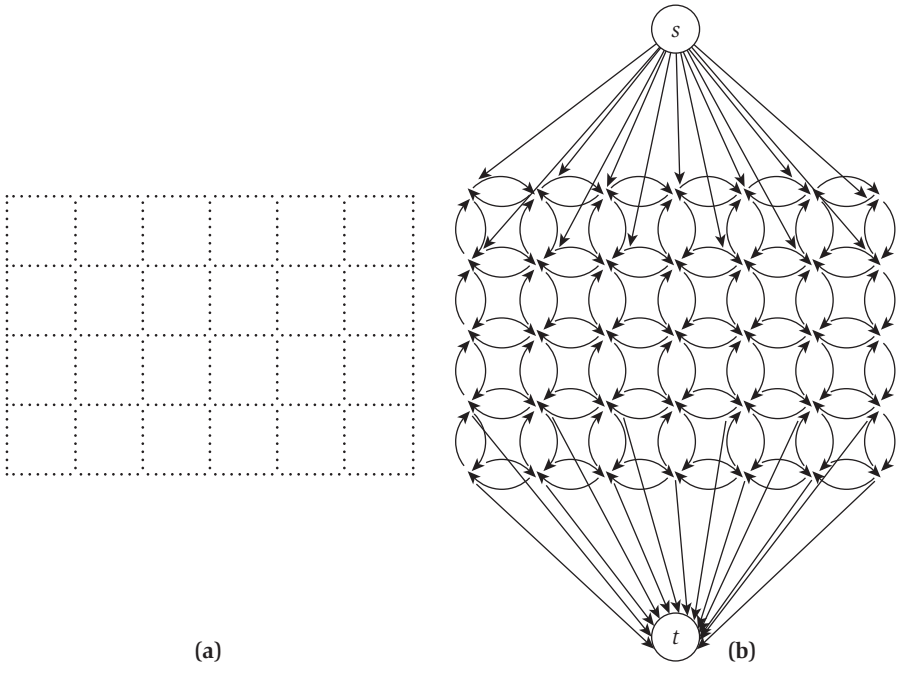

Figure 7.18 (a) A pixel graph. (b) A sketch of the corresponding flow graph. Not all edges from the source or to the sink are drawn.

For each pixel i, we have a *likelihood* a_i that it belongs to the foreground, and a *likelihood* b_i that it belongs to the background. For our purposes, we will assume that these likelihood values are arbitrary nonnegative numbers provided as part of the problem, and that they specify how desirable it is to have pixel i in the background or foreground. Beyond this, it is not crucial precisely what physical properties of the image they are measuring, or how they were determined.

In isolation, we would want to label pixel i as belonging to the foreground if $a_i > b_i$, and to the background otherwise. However, decisions that we make about the neighbors of i should affect our decision about i. If many of i's neighbors are labeled "background," for example, we should be more inclined to label i as "background" too; this makes the labeling "smoother" by minimizing the amount of foreground/background boundary. Thus, for each pair (i, j) of neighboring pixels, there is a *separation penalty* $p_{ij} \geq 0$ for placing one of i or j in the foreground and the other in the background.

We can now specify our *Segmentation Problem* precisely, in terms of the likelihood and separation parameters: It is to find a partition of the set of pixels into sets A and B (foreground and background, respectively) so as to maximize

$$q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}.$$

Thus we are rewarded for having high likelihood values and penalized for having neighboring pairs (i, j) with one pixel in A and the other in B. The problem, then, is to compute an *optimal labeling*—a partition (A, B) that maximizes $q(A, B)$.

Designing and Analyzing the Algorithm

We notice right away that there is clearly a resemblance between the minimum-cut problem and the problem of finding an optimal labeling. However, there are a few significant differences. First, we are seeking to maximize an objective function rather than minimizing one. Second, there is no source and sink in the labeling problem; and, moreover, we need to deal with values a_i and b_i on the nodes. Third, we have an undirected graph G, whereas for the minimum-cut problem we want to work with a directed graph. Let's address these problems in order.

We deal with the fact that our Segmentation Problem is a maximization problem through the following observation. Let $Q = \sum_i (a_i + b_i)$. The sum $\sum_{i \in A} a_i + \sum_{j \in B} b_j$ is the same as the sum $Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j$, so we can write

$$q(A, B) = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}.$$

Thus we see that the maximization of $q(A, B)$ is the same problem as the minimization of the quantity

$$q'(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}.$$

As for the missing source and the sink, we work by analogy with our constructions in previous sections: We create a new "super-source" s to represent the foreground, and a new "super-sink" t to represent the background. This also gives us a way to deal with the values a_i and b_i that reside at the nodes (whereas minimum cuts can only handle numbers associated with edges). Specifically, we will attach each of s and t to every pixel, and use a_i and b_i to define appropriate capacities on the edges between pixel i and the source and sink respectively.

Finally, to take care of the undirected edges, we model each neighboring pair (i, j) with *two* directed edges, (i, j) and (j, i), as we did in the undirected Disjoint Paths Problem. We will see that this works very well here too, since in any s-t cut, at most one of these two oppositely directed edges can cross from the s-side to the t-side of the cut (for if one does, then the other must go from the t-side to the s-side).

Specifically, we define the following flow network $G' = (V', E')$ shown in Figure 7.18(b). The node set V' consists of the set V of pixels, together with two additional nodes s and t. For each neighboring pair of pixels i and j, we add directed edges (i, j) and (j, i), each with capacity p_{ij}. For each pixel i, we add an edge (s, i) with capacity a_i and an edge (i, t) with capacity b_i.

Now, an s-t cut (A, B) corresponds to a partition of the pixels into sets A and B. Let's consider how the capacity of the cut $c(A, B)$ relates to the quantity $q'(A, B)$ that we are trying to minimize. We can group the edges that cross the cut (A, B) into three natural categories.

- Edges (s, j), where $j \in B$; this edge contributes a_j to the capacity of the cut.
- Edges (i, t), where $i \in A$; this edge contributes b_i to the capacity of the cut.
- Edges (i, j) where $i \in A$ and $j \in B$; this edge contributes p_{ij} to the capacity of the cut.

Figure 7.19 illustrates what each of these three kinds of edges looks like relative to a cut, on an example with four pixels.

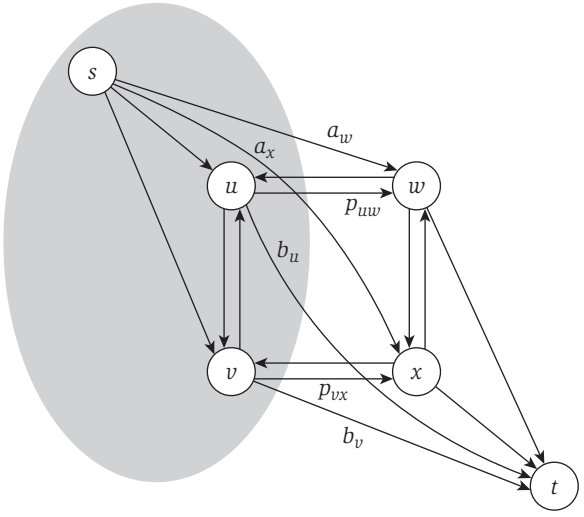

Figure 7.19 An s-t cut on a graph constructed from four pixels. Note how the three types of terms in the expression for $q'(A, B)$ are captured by the cut.

If we add up the contributions of these three kinds of edges, we get

$$c(A,B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}|=1}} p_{ij}$$
$$= q'(A,B).$$

So everything fits together perfectly. The flow network is set up so that the capacity of the cut (A, B) exactly measures the quantity $q'(A, B)$: The three kinds of edges crossing the cut (A, B), as we have just defined them (edges from the source, edges to the sink, and edges involving neither the source nor the sink), correspond to the three kinds of terms in the expression for $q'(A, B)$.

Thus, if we want to minimize $q'(A, B)$ (since we have argued earlier that this is equivalent to maximizing $q(A, B)$), we just have to find a cut of minimum capacity. And this latter problem, of course, is something that we know how to solve efficiently.

Thus, through solving this minimum-cut problem, we have an optimal algorithm in our model of foreground/background segmentation.

(7.55) *The solution to the Segmentation Problem can be obtained by a minimum-cut algorithm in the graph G' constructed above. For a minimum cut (A', B'), the partition (A, B) obtained by deleting s^* and t^* maximizes the segmentation value $q(A, B)$.*

7.11 Project Selection

Large (and small) companies are constantly faced with a balancing act between projects that can yield revenue, and the expenses needed for activities that can support these projects. Suppose, for example, that the telecommunications giant CluNet is assessing the pros and cons of a project to offer some new type of high-speed access service to residential customers. Marketing research shows that the service will yield a good amount of revenue, but it must be weighed against some costly preliminary projects that would be needed in order to make this service possible: increasing the fiber-optic capacity in the core of their network, and buying a newer generation of high-speed routers.

What makes these types of decisions particularly tricky is that they interact in complex ways: in isolation, the revenue from the high-speed access service might not be enough to justify modernizing the routers; *however*, once the company has modernized the routers, they'll also be in a position to pursue a lucrative additional project with their corporate customers; and maybe this additional project will tip the balance. And these interactions chain together: the corporate project actually would require another expense, but this in turn would enable two other lucrative projects—and so forth. In the end, the question is: Which projects should be pursued, and which should be passed up? It's a basic issue of balancing costs incurred with profitable opportunities that are made possible.

The Problem

Here's a very general framework for modeling a set of decisions such as this. There is an underlying set P of *projects*, and each project $i \in P$ has an associated *revenue* p_i, which can either be positive or negative. (In other words, each of the lucrative opportunities and costly infrastructure-building steps in our example above will be referred to as a separate project.) Certain projects are prerequisites for other projects, and we model this by an underlying directed acyclic graph $G = (P, E)$. The nodes of G are the projects, and there is an edge (i, j) to indicate that project i can only be selected if project j is selected as well. Note that a project i can have many prerequisites, and there can be many projects that have project j as one of their prerequisites. A set of projects $A \subseteq P$ is *feasible* if the prerequisite of every project in A also belongs to A: for each $i \in A$, and each edge $(i, j) \in E$, we also have $j \in A$. We will refer to requirements of this form as *precedence constraints*. The profit of a set of projects is defined to be

$$\texttt{profit}(A) = \sum_{i \in A} p_i.$$

The *Project Selection Problem* is to select a feasible set of projects with maximum profit.

This problem also became a hot topic of study in the mining literature, starting in the early 1960s; here it was called the *Open-Pit Mining Problem*.[3] Open-pit mining is a surface mining operation in which blocks of earth are extracted from the surface to retrieve the ore contained in them. Before the mining operation begins, the entire area is divided into a set P of *blocks*, and the net value p_i of each block is estimated: This is the value of the ore minus the processing costs, for this block considered in isolation. Some of these net values will be positive, others negative. The full set of blocks has precedence constraints that essentially prevent blocks from being extracted before others on top of them are extracted. The Open-Pit Mining Problem is to determine the most profitable set of blocks to extract, subject to the precedence constraints. This problem falls into the framework of project selection—each block corresponds to a separate project.

Designing the Algorithm

Here we will show that the Project Selection Problem can be solved by reducing it to a minimum-cut computation on an extended graph G', defined analogously to the graph we used in Section 7.10 for image segmentation. The idea is to construct G' from G in such a way that the source side of a minimum cut in G' will correspond to an optimal set of projects to select.

To form the graph G', we add a new source s and a new sink t to the graph G as shown in Figure 7.20. For each node $i \in P$ with $p_i > 0$, we add an edge (s, i) with capacity p_i. For each node $i \in P$ with $p_i < 0$, we add an edge (i, t) with capacity $-p_i$. We will set the capacities on the edges in G later. However, we can already see that the capacity of the cut $(\{s\}, P \cup \{t\})$ is $C = \sum_{i \in P: p_i > 0} p_i$, so the maximum-flow value in this network is at most C.

We want to ensure that if (A', B') is a minimum cut in this graph, then $A = A' - \{s\}$ obeys the precedence constraints; that is, if the node $i \in A$ has an edge $(i, j) \in E$, then we must have $j \in A$. The conceptually cleanest way to ensure this is to give each of the edges in G capacity of ∞. We haven't previously formalized what an infinite capacity would mean, but there is no problem in doing this: it is simply an edge for which the capacity condition imposes no upper bound at all. The algorithms of the previous sections, as well as the Max-Flow Min-Cut Theorem, carry over to handle infinite capacities. However, we can also avoid bringing in the notion of infinite capacities by

[3] In contrast to the field of data mining, which has motivated several of the problems we considered earlier, we're talking here about actual mining, where you dig things out of the ground.

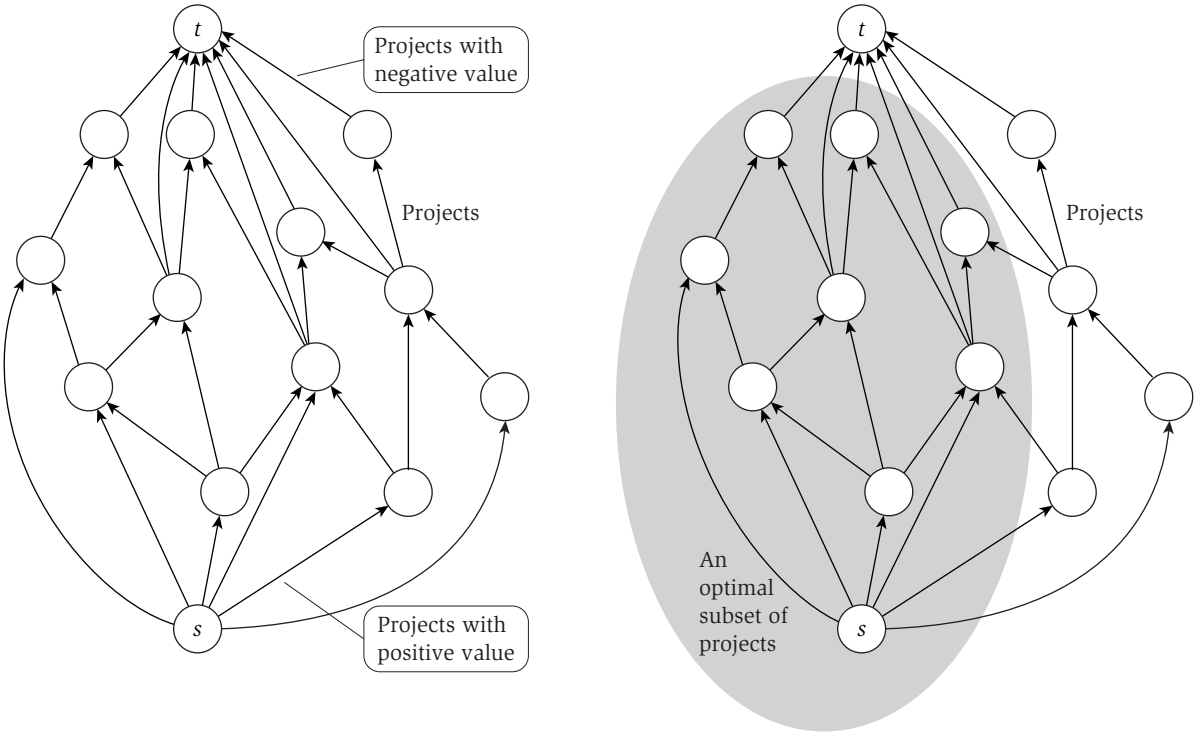

Figure 7.20 The flow graph used to solve the Project Selection Problem. A possible minimum-capacity cut is shown on the right.

simply assigning each of these edges a capacity that is "effectively infinite." In our context, giving each of these edges a capacity of $C + 1$ would accomplish this: The maximum possible flow value in G' is at most C, and so no minimum cut can contain an edge with capacity above C. In the description below, it will not matter which of these options we choose.

We can now state the algorithm: We compute a minimum cut (A', B') in G', and we declare $A' - \{s\}$ to be the optimal set of projects. We now turn to proving that this algorithm indeed gives the optimal solution.

Analyzing the Algorithm

First consider a set of projects A that satisfies the precedence constraints. Let $A' = A \cup \{s\}$ and $B' = (P - A) \cup \{t\}$, and consider the s-t cut (A', B'). If the set A satisfies the precedence constraints, then no edge $(i, j) \in E$ crosses this cut, as shown in Figure 7.20. The capacity of the cut can be expressed as follows.

(7.56) *The capacity of the cut (A', B'), as defined from a project set A satisfying the precedence constraints, is $c(A', B') = C - \sum_{i \in A} p_i$.*

Proof. Edges of G' can be divided into three categories: those corresponding to the edge set E of G, those leaving the source s, and those entering the sink t. Because A satisfies the precedence constraints, the edges in E do not cross the cut (A', B'), and hence do not contribute to its capacity. The edges entering the sink t contribute

$$\sum_{i \in A \text{ and } p_i < 0} -p_i$$

to the capacity of the cut, and the edges leaving the source s contribute

$$\sum_{i \notin A \text{ and } p_i > 0} p_i.$$

Using the definition of C, we can rewrite this latter quantity as $C - \sum_{i \in A \text{ and } p_i > 0} p_i$. The capacity of the cut (A', B') is the sum of these two terms, which is

$$\sum_{i \in A \text{ and } p_i < 0} (-p_i) + \left(C - \sum_{i \in A \text{ and } p_i > 0} p_i \right) = C - \sum_{i \in A} p_i,$$

as claimed. ■

Next, recall that edges of G have capacity more than $C = \sum_{i \in P : p_i > 0} p_i$, and so these edges cannot cross a cut of capacity at most C. This implies that such cuts define feasible sets of projects.

(7.57) *If (A', B') is a cut with capacity at most C, then the set $A = A' - \{s\}$ satisfies the precedence constraints.*

Now we can prove the main goal of our construction, that the minimum cut in G' determines the optimum set of projects. Putting the previous two claims together, we see that the cuts (A', B') of capacity at most C are in one-to-one correspondence with feasible sets of project $A = A' - \{s\}$. The capacity of such a cut (A', B') is

$$c(A', B') = C - \texttt{profit}(A).$$

The capacity value C is a constant, independent of the cut (A', B'), so the cut with minimum capacity corresponds to the set of projects A with maximum profit. We have therefore proved the following.

(7.58) *If (A', B') is a minimum cut in G' then the set $A = A' - \{s\}$ is an optimum solution to the Project Selection Problem.*