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.

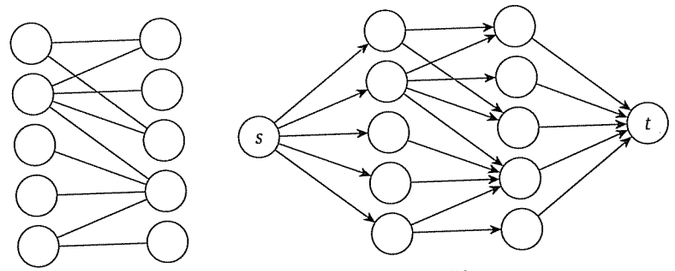

Figure 7.9 (a) A bipartite graph. (b) The corresponding flow network, with all capacities equal to 1.

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.

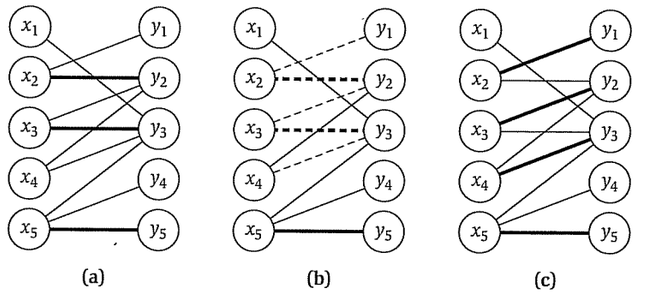

Figure 7.10 (a) A bipartite graph, with a matching M. (b) The augmenting path in the corresponding residual graph. (c) The matching obtained by the augmentation.

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

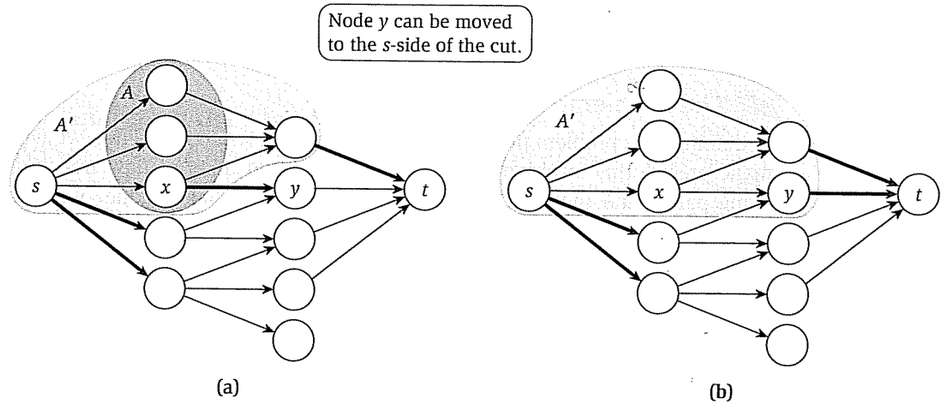

Figure 7.11 (a) A minimum cut in proof of (7.40). (b) The same cut after moving node y to the A' side. The edges crossing the cut are dark.

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 $|A| > |\Gamma(A)|$. ■

Referrence:

TextBook:

Algorithm Design, Jon Kleinberg and Eva Tardos, Pearson New International Edition.