U. Manber, "Introduction to Algorithms: A Creative Approach", Addison-Wesley, 1989, pp 234-243

true for problems that involve cycles. It is interesting to note that the problem of efficiently determining whether a directed graph contains an even-length cycle is still open (see the Bibliography section).

7.10 Matching

Given an undirected graph $G = (V, E)$, a **matching** is a set of edges no two of which have a vertex in common. The reason for the name is that an edge can be thought of as a match of its two vertices. We insist that no vertex belongs to more than one edge from the matching so that it is a monogamous matching. A vertex that is not incident to any edge in the matching is called **unmatched**. We also say that the vertex does not belong to the matching. A **perfect matching** is one in which all vertices are matched. A **maximum matching** is one with the maximum number of edges. A **maximal** matching, on the other hand, is a matching that cannot be extended by the addition of an edge. Problems involving matching occur in many situations (besides social). Workers may be matched to jobs, machines to parts, and so on. Furthermore, many problems that seem unrelated to matching have equivalent formulations in terms of matching problems.

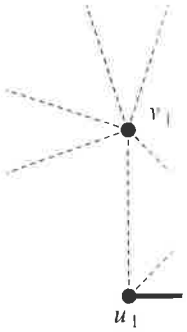
Matching in general graphs is a difficult problem. In this section, we limit our discussion to two specific matching problems. The first problem is not so important; it involves finding perfect matchings in special very dense graphs. The solution to this problem, however, illustrates an interesting approach, which we then generalize to solve an important problem concerning matching in bipartite graphs.

7.10.1 Perfect Matching in Very Dense Graphs

In this example, we consider a very restricted case of the perfect matching problem. Let $G = (V, E)$ be an undirected graph such that $|V| = 2n$ and the degree of each vertex is at least n. We present an algorithm to find a perfect matching in such graphs. As a corollary, we show that, under these conditions, a perfect matching always exists.

We use induction on the size m of the matching. The base case, $m = 1$, is handled by taking any arbitrary edge as a matching of size one. We will show that we can extend any matching that is not perfect either by adding another edge or by replacing an existing edge with two new edges. In either case, the size of the matching is increased, and the result follows.

Consider a matching M in G with m edges such that $m < n$. We first check all the edges not in M to see whether any of them can be added to M. If we find such an edge, then we are done. Otherwise, M is a maximal matching. Since M is not perfect, there are at least two nonadjacent vertices, v_1 and v_2, that do not belong M. These two vertices have at least $2n$ distinct edges coming out of them. All of these edges lead to vertices that are covered by M, since otherwise such an edge could be added to M. Since the number of edges in M is $< n$ and there are $2n$ edges from v_1 and v_2 adjacent to them, at least one edge from M — say (u_1, u_2) — is adjacent to three edges from v_1 and v_2. Assume, without loss of generality, that those three edges are (u_1, v_1), (u_1, v_2), and (u_2, v_1) (see Fig. 7.36(a)). It is easy to see that, by removing the edge (u_1, u_2) from M



and adding the t
We leave
algorithm is anc
in each step in t
case, but, in ge
may affect cho
generalize this a

7.10.2 Bipa

Let $G = (V, E, l$
and E is a set of

The Pr
graph G.

We can formula
boys, and E is
maximize the nu
A straight
more matches a
least come clos
approach by firs
will be more lik
boys that are th
to analyze such
problem. Supp
maximum matc

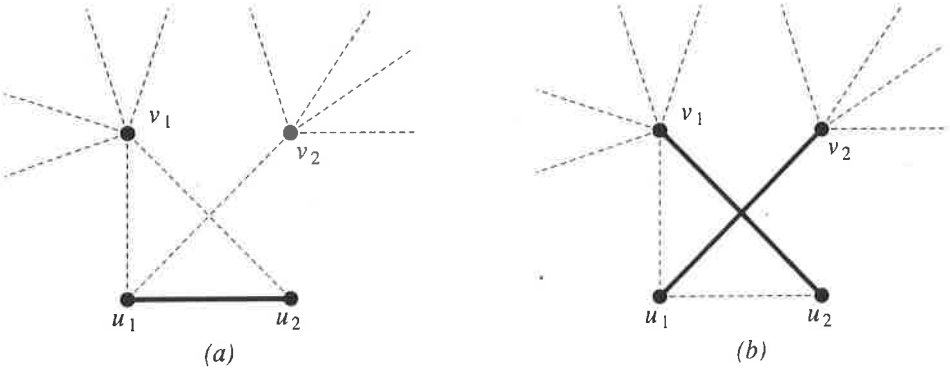


Figure 7.36 Extending a matching.

and adding the two edges (u_1, v_2), and (u_2, v_1), we get a larger matching (Fig. 7.36(b)).

We leave the implementation of this algorithm as an exercise (7.21). This algorithm is another example of a *greedy* approach. At most three edges were involved in each step in the extension of one matching to a larger one. This was sufficient in this case, but, in general, finding a good matching is more difficult. A choice of one edge may affect choices of other edges far away in the graph. Next, we show how to generalize this approach to other matching problems.

7.10.2 Bipartite Matching

Let $G = (V, E, U)$ be a bipartite graph, such that V and U are two disjoint sets of vertices, and E is a set of edges connecting vertices from V to vertices in U.

> **The Problem** Find a maximum-cardinality matching in a bipartite graph G.

We can formulate this problem in terms of real matching: V is a set of girls, U is a set of boys, and E is a set of "possible" pairings; we want to match boys to girls so as to maximize the number of matched boys and girls.

A straightforward approach is to try to match according to some strategy until no more matches are possible, in the hope that the strategy will guarantee optimality, or at least come close. We can try different strategies. For example, we can try a greedy approach by first matching the vertices with small degrees, hoping that the other vertices will be more likely to have unmatched partners later on. (In other words, first match the boys that are the most difficult to match, and worry about the rest later.) Instead of trying to analyze such strategies (which is hard), we try the approach used in the previous problem. Suppose that we start with a maximal matching, which is not necessarily a maximum matching. Can we somehow improve it? Consider Fig. 7.37(a), in which the

matching is depicted by bold lines. It is clear that we can improve the matching by replacing the edge *2A* with the edges *1A* and *2B*. This is similar to the transformation we applied in the previous problem. But we are not restricted to replacing one edge with two edges. If we find a similar situation where k edges can be replaced by $k+1$ edges, then we have an improvement. For example, we can improve the matching further by replacing the edges *3D* and *4E* with the edges *3C*, *4D*, and *5E* (Fig. 7.37(b)).

Let's study these transformations. Our goal is to add more matched vertices. We start with an unmatched vertex v and try to find a match for it. If we already have a maximal matching, then all of v's neighbors are already matched, so we must try to break up a match. We choose another vertex u, adjacent to v, which was previously matched to, say, w. We match v to u and break up the match between u and w. We now have to find a match for w. If w is connected to an unmatched vertex, then we are done (this was the first case above); if not, we can continue this way by breaking matches and trying rematches. To translate this attempt into an algorithm, we have to do two things. First, we have to make sure that this procedure terminates, and second, we have to show that, if there is an improvement, then this procedure will find it. First, we formalize this idea.

An **alternating path** P for a given matching M is a path from a vertex v in V to a vertex u in U, both of which are unmatched in M, such that the edges of P are alternatively in $E-M$ and in M. That is, the first edge (v, w) of P does not belong to M (since v does not belong to M), the second edge (w, x) belongs to M, and so on, until the last edge of P, (z, u), which does not belong to M. Notice that alternating paths are exactly what we used already to improve a matching. The number of edges in P must be odd since P starts in V and ends in U. Furthermore, there is exactly one more edge of P in $E-M$ than there is in M. Therefore, if we replace all the edges of P that belong to M by the edges that do not belong to M, we get another matching with one more edge. For example, the first alternating path we used to improve the matching in Fig. 7.37(a) was (*1A*, *A2*, *2B*), which was used to replace the edge *A2* with the edges *1A* and *2B*; the second alternating path was (*C3*, *3D*, *D4*, *4E*, *E5*), which was used to replace the edges *3D* and *4E* with the edges *C3*, *D4*, and *E5*.

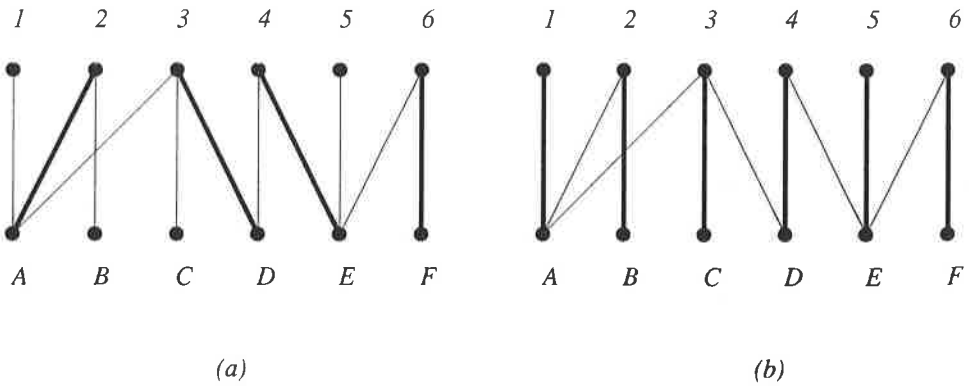
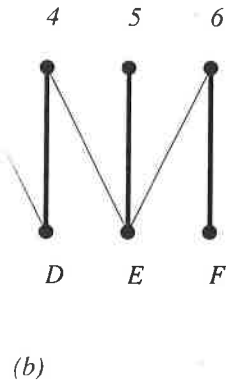


Figure 7.37 Extending a bipartite matching.

ove the matching by
he transformation we
ng one edge with two
by $k+1$ edges, then
matching further by
.37(b)).
atched vertices. We
f we already have a
we must try to break
previously matched
w. We now have to
e are done (this was
matches and trying
lo two things. First,
have to show that, if
rmalize this idea.
a vertex v in V to a
he edges of P are
oes not belong to M
. and so on, until the
lternating paths are
edges in P must be
one more edge of P
P that belong to M
one more edge. For
in Fig. 7.37(a) was
ges *1A* and *2B*; the
to replace the edges



(b)

It should be clear now that, if there is an alternating path for a given matching M, then M is not maximum. It turns out that the opposite is also true.

□ Alternating-Path Theorem

A matching is maximum if and only if it has no alternating paths. □

This claim will be proved, in the context of a more general theorem, in the next section.

The alternating path theorem immediately suggests an algorithm, because any matching that is not maximum has an alternating path and any alternating path can extend a matching. We start with the greedy algorithm, adding as many edges to the matching as possible, until we get a maximal matching. We then search for an alternating path, and modify the matching accordingly until no more alternating paths can be found. The resulting matching is maximum. Since each alternating path extends a matching by one edge and there are at most $n/2$ edges in any matching (where n is the number of vertices), the number of iterations is at most $n/2$. The only remaining problem is how to find alternating paths. We solve this problem as follows. We transform the undirected graph G to a directed graph G' by directing the edges in M to point from U to V and directing the edges not in M to point from V to U. Figure 7.38(a) shows the matching obtained for the graph in Fig. 7.37(a), and Fig. 7.38(b) shows the directed graph G'. An alternating path corresponds exactly to a directed path from an unmatched vertex in V to an unmatched vertex in U. Such a directed path can be found by any graph-search procedure, for example, DFS. The complexity of a search is $O(|V| + |E|)$; hence, the complexity of the algorithm is $O(|V|(|V| + |E|))$.

An Improvement

Since a search can traverse the whole graph in the same worst-case running time that it traverses one path, we might as well try to find several alternating paths with one search. We have to make sure, however, that these paths do not modify one another. One way to guarantee the independence of such alternating paths is to restrict them to be vertex

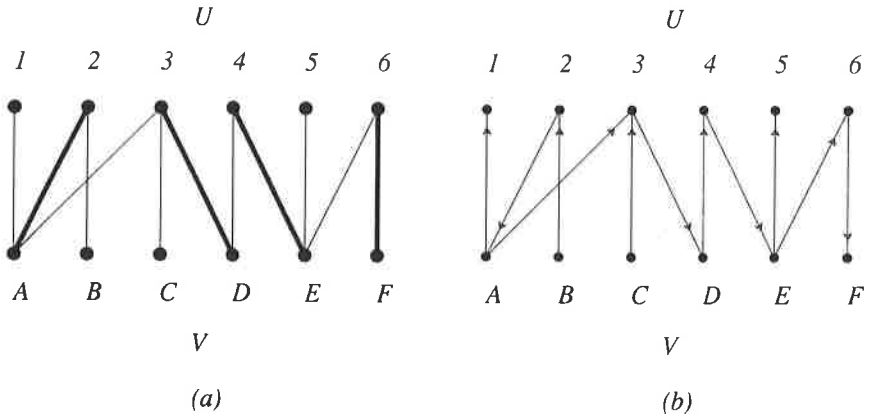


Figure 7.38 Finding alternating paths.

disjoint. If the paths are vertex disjoint, they modify different vertices, so they can be applied concurrently. The new improved algorithm for finding alternating paths is the following. First, we perform BFS in G' from the set of all unmatched vertices in V, level by level, until a level in which unmatched vertices in U are found. Then, we extract from the graph induced by the BFS a *maximal* set of vertex disjoint paths in G' (which are alternating paths in G). This is done by finding any path, removing its vertices, finding another path, removing its vertices, and so on. (The result is not a *maximum* set, but merely a maximal set.) We choose a maximal set in order to maximize the number of edges added to the matching with one search (each vertex-disjoint alternating path adds one edge to the matching). Finally, we modify the matching using this set of alternating paths. This process is repeated until no more alternating paths can be found (i.e., the new directed graph G' disconnects the unmatched vertices in V from the unmatched vertices in U).

Complexity It turns out that the number of iterations of the improved algorithm is $O(\sqrt{|V|})$ in the worst case. We omit the proof, which is due to Hopcroft and Karp [1973]. The overall worst-case running time is thus $O((|V| + |E|)\sqrt{|V|})$.

7.11 Network Flows

The problem of network flows is a basic problem in graph theory and combinatorial optimization. It has been studied extensively for the last 35 years, and many algorithms and data structures have been developed for it. It has many variations and extensions. Furthermore, many seemingly unrelated problems can be posed as network-flow problems. The basic variation of the network-flow problem is defined as follows. Let $G = (V, E)$ be a directed graph with two distinguished vertices, s (the source) with indegree 0, and t (the sink) with outdegree 0. Each edge e in E has an associated positive weight $c(e)$, called the **capacity** of e. The capacity measures the amount of flow that can pass through an edge. We call such a graph a **network**. For convenience we assign a capacity of 0 to nonexisting edges. A **flow** is a function f on the edges of the network that satisfies the following two conditions:

1. $0 \le f(e) \le c(e)$: The flow through an edge cannot exceed the capacity of that edge.
2. For all $v \in V - \{s, t\}$, $\sum_u f(u, v) = \sum_w f(v, w)$: The total flow entering a vertex is equal to the total flow exiting this vertex (except for the source and sink).

These two conditions imply that the total flow leaving s is equal to the total flow entering t. The problem is to maximize this flow. (If the capacities are real numbers, then it is not even clear that maximum flows exist; we will show that they indeed always exist.) One way to visualize this problem is to think of the network as a network of water pipes. The goal is to push as much water through the pipes as possible. If too much water is pushed to the wrong area, the pipes will burst.

First, we show t
section, can be posed
exercise, since we alre
how to solve the netwo
The reason we present
network-flow problem
Understanding the simi

Given a bipartit
cardinality matching, w
connect all vertices in
7.39, in which all edge
all the edges, and we h
M be a matching in G
We assign a flow of
matched vertices in M
thus equal to the num
matching if and only
clear: If the flow is m
larger matching, since
claim we somehow ha
show that, if there are
proceed to do just that.

An **augmenting**
which consists of edge
edges (v, u) satisfies e

1. (v, u) is in the
case, the edge (

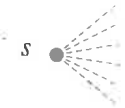


Figure 7.39 Reducing
edges are from left to

First, we show that the problem of bipartite matching, discussed in the previous section, can be posed as a network-flow problem. This may seem to be a fruitless exercise, since we already know how to solve the matching problem, but we do not know how to solve the network-flow problem (namely, the reduction is in the wrong direction). The reason we present this wrong-order reduction is that the techniques for solving the network-flow problem are similar to those for solving the bipartite matching problem. Understanding the similarities can be helpful in understanding network-flow algorithms.

Given a bipartite graph $G=(V, E, U)$ in which we want to find a maximum-cardinality matching, we add two new vertices s and t, connect s to all vertices in V, and connect all vertices in U to t. We also direct all the edges in E from V to U (see Fig. 7.39, in which all edges are directed from left to right). We now assign capacities of 1 to all the edges, and we have a valid network-flow problem on the modified graph G'. Let M be a matching in G. There is a natural correspondence between M and a flow in G'. We assign a flow of 1 to all the edges in M and to all the edges connecting s or t to matched vertices in M. All the other edges are assigned a flow of 0. The total flow is thus equal to the number of edges in the matching. It turns out that M is a maximum matching if and only if the corresponding flow is a maximum flow in G'. One side is clear: If the flow is maximum and it corresponds to a matching, then we cannot have a larger matching, since it would correspond to a larger flow. For the other side of the claim we somehow have to adapt the idea of alternating paths to network flows, and to show that, if there are no alternating paths, then the corresponding flow is maximum. We proceed to do just that.

An **augmenting path** with respect to a given flow f is a directed path from s to t which consists of edges from G, but not necessarily in the same direction; each of these edges (v, u) satisfies exactly one of the following two conditions:

1. (v, u) is in the same direction as it is in G, and $f(v, u) < c(v, u)$. In this case, the edge (v, u) is called a **forward edge**. A forward edge has room for

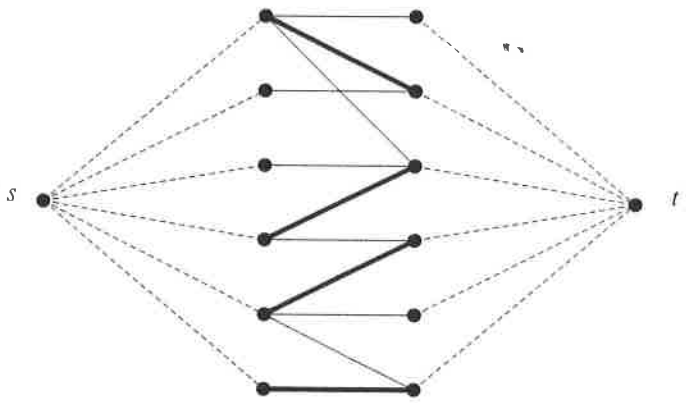


Figure 7.39 Reducing bipartite matching to network flow (the directions of all the edges are from left to right).

more flow. The difference $c(v, u) - f(v, u)$ is called the **slack** of the edge.

2. (v, u) is in the opposite direction in G (namely, $(u, v) \in E$), and $f(u, v) > 0$. In this case, the edge (v, u) is called a **backward edge**. It is possible to borrow some flow from a backward edge.

Augmenting paths are extensions of alternating paths, and they serve the same purpose for network flows as alternating paths do for bipartite matching. If there exists an augmenting path with respect to a flow f (we say that f **admits** an augmenting path), then f is not maximum. We can modify f by moving more flow through the augmenting path in the following way. If all the edges of the path are forward edges, then more flow can be moved through them, and all the constraints are still satisfied. The extra flow in that case is exactly the minimum slack of the edges in the path. The case of backward edges is a little more complicated. Consider Fig. 7.40. Each edge is marked with two numbers a/b, such that a is the capacity and b is the current flow. It is clear that no more flow can be pushed forward, since there is no path from s to t that consists of only forward edges. However, there is a way to extend the flow.

The path $s-v-u-w-t$ is an augmenting path. An additional flow of 2 can reach u from s through this path (2 is the minimum slack over all forward edges until u). We can *deduct* a flow of 2 from $f(w, u)$. The conservation constraint is now satisfied for u, since u had an additional flow of 2 coming in through the augmenting path, and a flow of 2 deducted from the backward edge. We now have an extra flow of 2 at w that needs to be pushed, which is exactly what we want. We can continue pushing flow from w in the same way, pushing it forward on forward edges, and deducting it from backward edges. In this case, there is one forward edge (w, t) that reaches t, and we are done. Since only forward edges can leave s and enter t, the total flow is increased. The increase is equal to the minimum of either the minimal slack of forward edges or the minimal current flow through backward edges. Figure 7.41 shows the same network with the modified flow. (This flow is in fact maximum.)

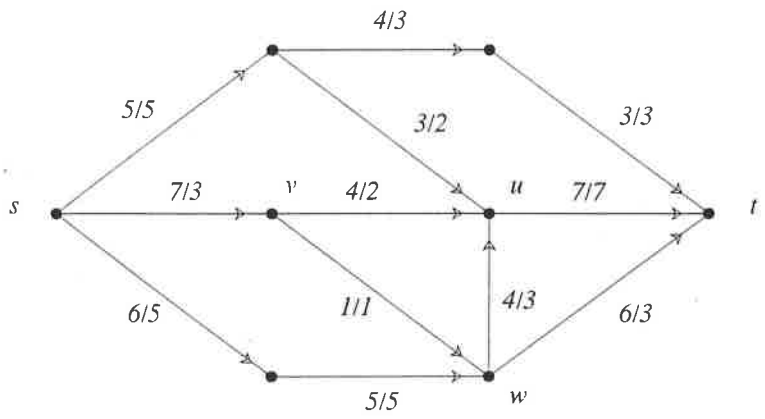


Figure 7.40 An example of a network with a (nonmaximum) flow.

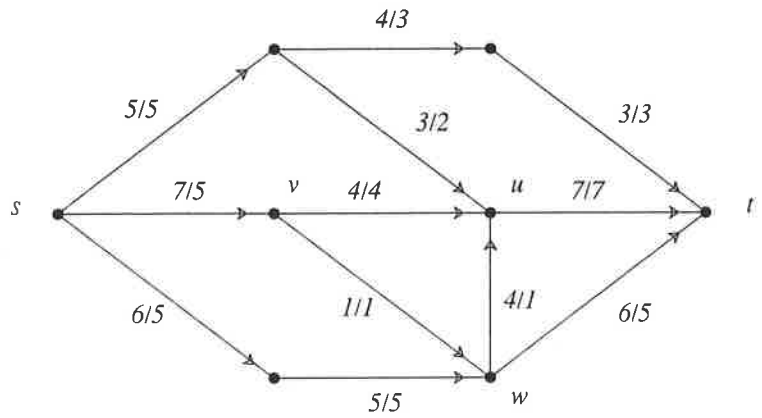


Figure 7.41 The result of augmenting the flow of Fig. 7.40.

The arguments above establish that if there is an augmenting path, then the flow is not maximum. The opposite is also true:

□ The Augmenting-Path Theorem

A flow f is maximum if and only if it admits no augmenting path.

Proof: We have already shown one direction of the theorem — if the flow admits an augmenting path, then it is not maximum. Let's assume now that a flow f admits no augmenting path, and prove that f is maximum. We use the concept of **cuts**. Intuitively, a cut is a set of edges that separate s from t. More precisely, let A be a set of vertices of V such that $s \in A$ and $t \notin A$. Denote the rest of the vertices by $B = V - A$. A cut is the set of edges $\{(v, w) \in E\}$ such that $v \in A$ and $w \in B$. The capacity of the cut is defined as the sum of the capacities of its edges. It is clear that no flow can exceed the capacity of any cut. (If you disconnect the pipes, no water can flow through them.) Hence, if we find a flow whose value is equal to the capacity of a (any) cut, then this flow must be maximum. We proceed to prove that, if a flow admits no augmenting paths, then it is equal to the capacity of a cut, and hence it is maximum.

Let f be a flow that admits an augmenting path. Let $A \subset V$ be the set of vertices such that for each $v \in A$ there is an augmenting path, with respect to the flow f, from s to v. Clearly, $s \in A$, and $t \notin A$ (since we assumed that f admits no augmenting path). Therefore, A defines a cut. We claim that, for all edges (v, w) in that cut, $f(v, w) = c(v, w)$. Otherwise, (v, w) would be a forward edge and there would be an augmenting path to w, contrary to our assumption that $w \notin A$. By the same argument, there cannot be an edge (w, v) such that $w \notin A$ and $v \in A$, and $f(w, v) > 0$ (since it would be a backward edge and it could extend an augmenting path). Hence, the value of the flow f is equal to the capacity of the cut defined by A, and f is maximum. □

We have proved the following fundamental theorem.

□ Max-Flow Min-Cut Theorem

The value of a maximum flow in a network is equal to the minimum capacity of a cut. □

The augmenting-path theorem also implies the following theorem.

□ The Integral-Flow Theorem

If the capacities of all edges in the network are integers, then there is a maximum flow whose value is an integer.

Proof: The theorem follows directly from the augmenting-path theorem. In fact, any algorithm that uses only augmenting paths will lead to an integral flow if all the capacities are integers. This is obvious since we start with a flow of 0, and each augmenting path adds an integer to the total flow. □

We now return to the bipartite-matching problem. Clearly, any alternating path in G corresponds to an augmenting path in G', and vice versa. The augmenting-path theorem implies the alternating-path theorem given in the previous section. If M is a maximum matching, then there is no alternating path for it, which implies that there is no augmenting path in G', which implies that the flow is maximum. On the other hand, there is a maximum integral flow, and it clearly corresponds to a matching since each vertex in V is connected by only one edge (with capacity 1) to s; hence, each vertex of V can support a flow of only 1. The same argument holds for the vertices of U. This matching must be maximum since, if it could be extended, then there would be a larger flow.

The augmenting-path theorem immediately suggests an algorithm. We start with a flow of 0, search for augmenting paths, and augment the flow accordingly, until there are no more augmenting paths. We are always making progress since we are increasing the flow. Searching for augmenting paths can be done in the following way. We define the **residual graph**, with respect to a network $G=(V, E)$ and a flow f, as the network $R=(V, F)$ with the same vertices, the same source and sink, and the same edges, but with possibly different directions and different capacities. The edges in the residual graph correspond to the possible edges in an augmenting path. Their capacities correspond to the possible augmenting flow through those edges. More precisely, an edge (v, w) belongs to F if it is either a forward edge, in which case its capacity is $c(v, w)-f(v, w)$, or a backward edge, in which case its capacity is $f(v, w)$. An augmenting path is thus a regular directed path from s to t in the residual graph. Constructing the residual graph requires $|E|$ steps since each edge has to be checked exactly once.

Unfortunately, selecting augmenting paths in an arbitrary way may lead to a very slow algorithm. The worst-case running time of such an algorithm may not even be a function of the size of the graph. Consider the network in Fig. 7.42. The maximum flow is obviously $2M$. However, one might start with the path $s-a-b-t$, which can support a flow of only 1. Then, one might take the augmenting path $s-b-a-t$, which again

augments the flow i
very large, even tho
of M can be repres
case, in the size of t
Although the
it. Furthermore, we
the algorithm. Edm
selecting the next a
number of edges.
$(|V|^3-|V|)/4$ aug
case is polynomial
suggested since then.
simple). An upper
achieved by several
(references are given

7.12 Hamilto

We started this chapt
end the chapter with
also a famous proble
who designed a popul

The Proble
that includes e

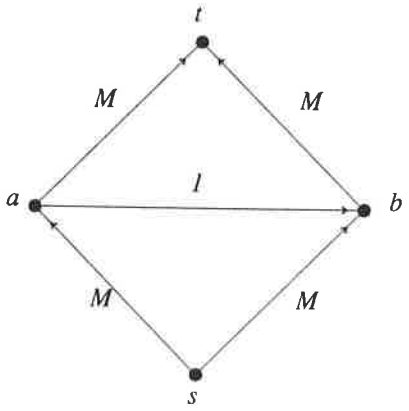


Figure 7.42 A bad example of network flow.

augments the flow by only 1. This process can be repeated $2M$ times, where M may be very large, even though the graph has only four vertices and five edges. (Since the value of M can be represented by $O(\log M)$ bits, this algorithm is exponential, in the worst case, in the size of the input.)

Although the scenario above may be unlikely, we have to take precautions to avoid it. Furthermore, we want to minimize the number of augmentations in order to speed up the algorithm. Edmonds and Karp [1972], for example, suggested (among other things) selecting the next augmenting path by taking the augmenting path with the minimum number of edges. They proved that, if this policy is maintained, then at most $(|V|^3 - |V|)/4$ augmentations are required. This leads to an algorithm whose worst case is polynomial in the size of the input. Many different algorithms have been suggested since then. Some are complicated; others are relatively simple (none are really simple). An upper bound of $O(|V|^3)$ on the complexity of network flow has been achieved by several of these algorithms. We will not describe these algorithms here (references are given in the Bibliography section).

7.12 Hamiltonian Tours

We started this chapter with a discussion of a tour containing all edges of a graph. We end the chapter with a discussion of a tour containing all the vertices of a graph. This is also a famous problem, named after the Irish mathematician Sir William R. Hamilton, who designed a popular game based on this problem in 1857.

The Problem Given a graph $G=(V, E)$, find a simple cycle in G that includes every vertex of V exactly once.