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 to 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

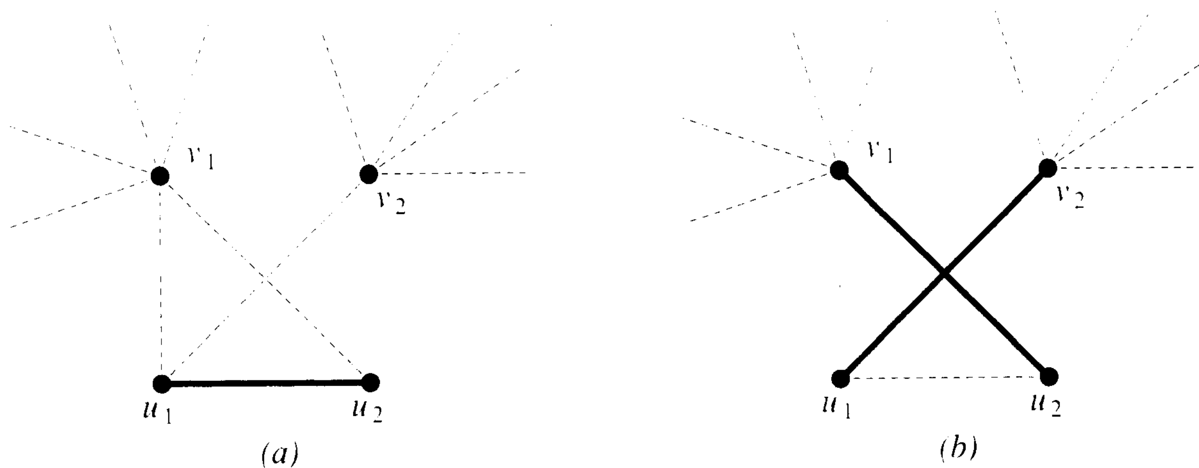

Figure 7.36 Extending a matching.

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*.

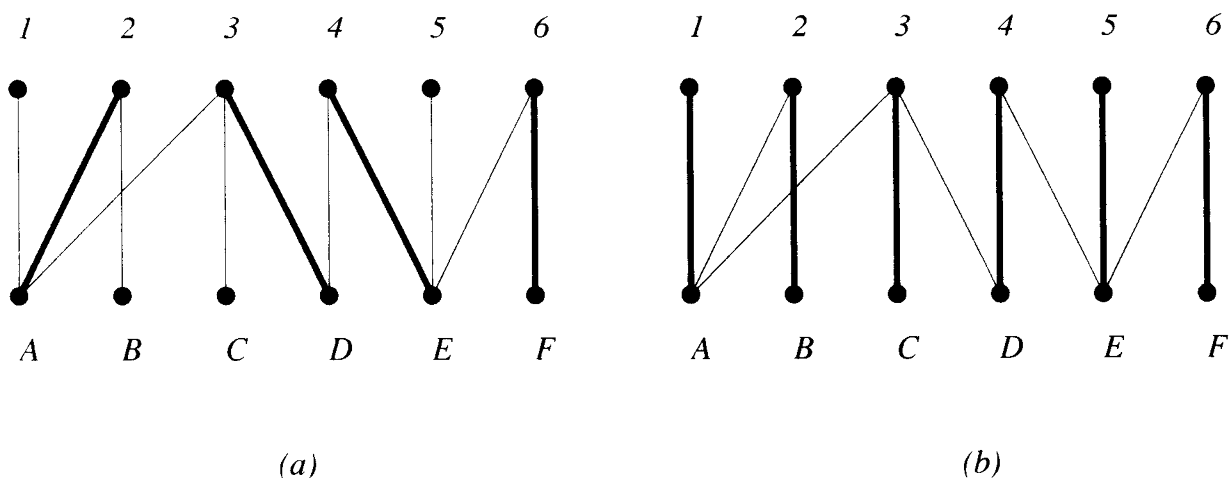

Figure 7.37 Extending a bipartite matching.

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

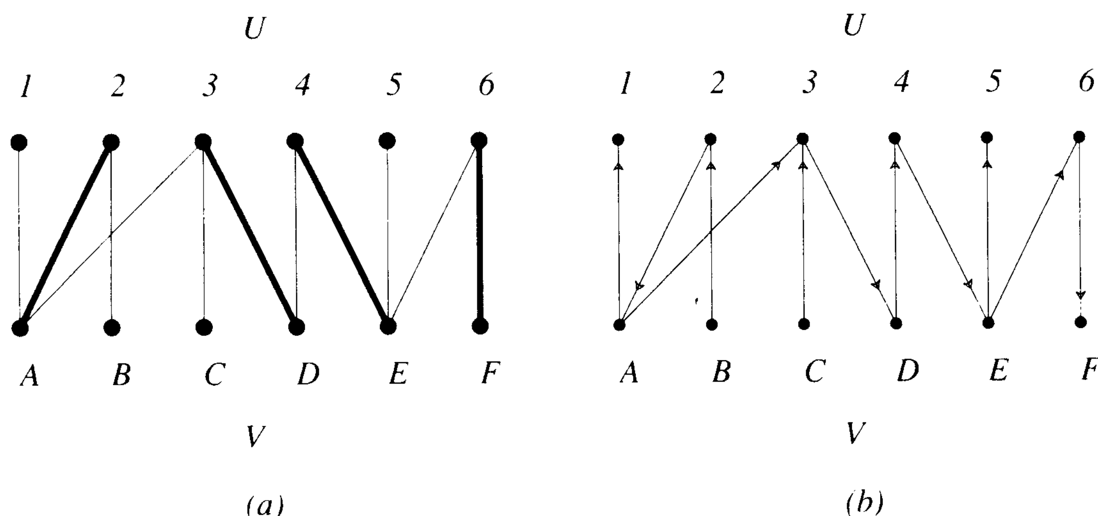

Figure 7.38 Finding alternating paths.

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.