CHAPTER 9

Matchings

9.1. Bipartite matchings and network flows. The last network optimization problem we shall study has close connections with the maximum flow problem. Let $G = [V, E]$ be an undirected graph each of whose edges has a real-valued *weight*, denoted by *weight* (v, w). A *matching* M on G is a set of edges no two of which have a common vertex. The *size* $|M|$ of M is the number of edges it contains; the *weight* of M is the sum of its edge weights. The *maximum matching problem* is that of finding a matching of maximum size or weight. We shall distinguish four versions of this problem, depending on whether we want to maximize size or weight (unweighted versus weighted matching) and whether G is bipartite or not (bipartite versus nonbipartite matching). The weighted bipartite matching problem is commonly called the *assignment problem;* one application is the assignment of people to tasks, where the weight of an edge $\{x, y\}$ represents the benefit of assigning person x to task y.

Bipartite matching problems can be viewed as a special case of network flow problems [8]. Suppose G is bipartite, with a vertex partition X, Y such that every edge has one end in X and the other in Y. We shall denote a typical edge by $\{x, y\}$ with $x \in X$ and $y \in Y$. Let s and t be two new vertices. Construct a graph G' with vertex set $V \cup \{s, t\}$, source s, sink t, and capacity-one edges $[s, x]$ of cost zero for every $x \in X$, $[y, t]$ of cost zero for every $y \in Y$, and $[x, y]$ of cost $-weight\ (x, y)$ for every $\{x, y\} \in E$. (See Fig. 9.1.) G' is a unit network as defined in Chapter 8.

An integral flow f on G' defines a matching on G of size $|f|$ and weight $-cost(f)$ given by the set of edges $\{x\ y\}$ such that $[x, y]$ has flow one. Conversely a matching M on G defines a flow of value $|M|$ and cost $-weight(M)$ that is one on each path $[s, x]$, $[x, y]$, $[y, t]$ such that $\{x, y\} \in M$. This means that we can solve a matching problem on G by solving a flow problem on G'.

Suppose we want a maximum size matching. Any integral maximum flow on G' gives a maximum size matching on G. We can find such a flow in $O(\sqrt{n}\, m)$ time using Dinic's algorithm, since G' is unit (see Theorem 8.8). Thus we have an $O(\sqrt{n}\, m)$-time algorithm for unweighted bipartite matching. This algorithm can be translated into the terminology of alternating paths (which we shall develop in the next section), and it was originally discovered in this form by Hopcroft and Karp [13]. Even and Tarjan [7] noted the connection with Dinic's algorithm.

Suppose we want a maximum weight matching. Since G' is acyclic, it has no negative cost cycles, and we can apply minimum cost augmentation to G' (see §8.4). Starting with the zero flow, this method will produce a sequence of at most $n/2$ minimum cost flows of increasing value, the last of which is a minimum cost maximum flow. Because successive augmenting paths have nondecreasing cost (Lemma 8.4), if we stop the algorithm just after the last augmentation along a path

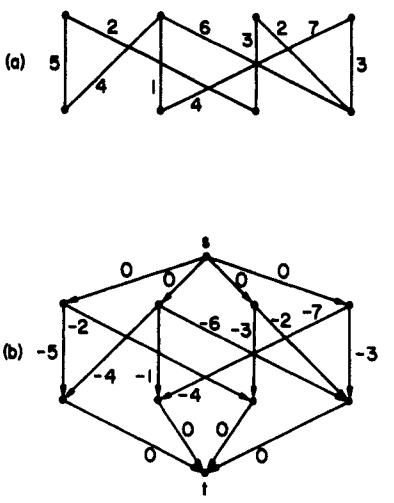


FIG. 9.1. *Transformation of a bipartite matching problem to a network flow problem.* (a) *Bipartite graph defining a weighted matching problem.* (b) *Corresponding network. Numbers on edges are costs; all capacities are one.*

of negative cost we will have a flow corresponding to a maximum weight matching. As implemented in §8.4, minimum cost augmentation solves the weighted bipartite matching problem in $O(n\, m \log_{(2+m/n)} n)$ time. This method, too, can be translated into the terminology of alternating paths, and it was discovered in this form by Kuhn [17], who named it the *Hungarian method* in recognition of Konig [15], [16] and Egervary's [5] work on maximum matching, which among other results produced the *Konig–Egervary theorem.* This theorem is the special case of the max-flow min-cut theorem for unweighted bipartite matching: the maximum size of a bipartite matching is equal to the minimum size of a vertex set containing at least one vertex of every edge.

9.2. Alternating paths. Nonbipartite matching is more complicated than bipartite matching. The idea of augmenting paths carries over from network flow theory, but to get efficient algorithms we need another idea, contributed by Edmonds in a paper with a flowery title [3]. In this section we shall develop the properties of augmenting paths in the setting of matching theory.

Let M be a matching. An edge in M is a *matching edge*; every edge not in M is *free.* A vertex is *matched* if it is incident to a matching edge and *free* otherwise. An *alternating path* or *cycle* is a simple path or cycle whose edges are alternately matching and free. The *length* of an alternating path or cycle is the number of edges it contains; its *weight* is the total weight of its free edges minus the weight of its matching edges. An alternating path is *augmenting* if both its ends are free vertices. If M has an augmenting path then M is not of maximum size, since we can increase its size by one by interchanging matching and free edges along the path. We call this an *augmentation.* The following theorem is analogous to Lemma 8.2:

THEOREM 9.1 [3, 13]. *Let M be a matching, $\overline{M}$ a matching of maximum size, and $k = |\overline{M}| - |M|$. Then M has a set of k vertex-disjoint augmenting paths, at least one of length at most $n/k - 1$.*

Proof. Let $M \oplus \overline{M}$ be the *symmetric difference* of M and $\overline{M}$, the set of edges in M or in $\overline{M}$ but not in both. Every vertex is adjacent to at most two edges of $M \oplus \overline{M}$; thus the subgraph of G induced by $M \oplus \overline{M}$ consists of a collection of paths and even-length cycles that are alternating with respect to M (and to $\overline{M}$). $M \oplus \overline{M}$ contains exactly k more edges in $\overline{M}$ than in M; thus it contains at least k paths that begin and end with an edge of $\overline{M}$. These paths are vertex-disjoint and augmenting for M; at least one has length at most $n/k - 1$. □

Berge [2] and Norman and Rabin [20] proved a weaker form of Theorem 9.1: A matching is of maximum size if and only if it has no augmenting path. We can construct a maximum size matching by beginning with the empty matching and repeatedly performing augmentations until there are no augmenting paths; this takes at most $n/2$ augmentations. We call this the *augmenting path method* for *maximum matching*. Before discussing how to find augmenting paths, let us obtain a result for weighted matchings analogous to Theorem 8.12.

THEOREM 9.2. *Let M be a matching of maximum weight among matchings of size $|M|$, let p be an augmenting path for M of maximum weight, and let M' be the matching formed by augmenting M using p. Then M' is of maximum weight among matchings of size $|M| + 1$.*

Proof. Let $\overline{M}$ be a matching of maximum weight among matchings of size $|M| + 1$. Consider the symmetric difference $M \oplus \overline{M}$. Define the weight of a path or cycle in $M \oplus \overline{M}$ with respect to M. Any cycle or even-length path in $M \oplus \overline{M}$ must have weight zero; a cycle or path of positive or negative weight contradicts the choice of M or $\overline{M}$, respectively. $M \oplus \overline{M}$ contains exactly one more edge in $\overline{M}$ than in M; thus we can pair all but one of the odd-length paths so that each pair has an equal number of edges in M and in $\overline{M}$. Each pair of paths must have total weight zero; a positive or negative weight pair contradicts the choice of M or $\overline{M}$. Augmenting M using the remaining path gives a matching of size $|M| + 1$ and of the same weight as $\overline{M}$. The theorem follows. □

Theorem 9.2 implies that the augmenting path method will compute maximum weight matchings of all possible sizes if we always augment using a maximum weight augmenting path. The analogue of Lemma 8.4 holds for matchings; namely, this method will augment along paths of successively decreasing weight. Thus if we want a maximum weight matching, we can stop after the last augmentation along a path of positive weight.

9.3. Blossoms. There remains the problem of finding augmenting paths, maximum weight or otherwise. The natural way to find an augmenting path is to search from the free vertices, advancing only along alternating paths. If a search from one free vertex reaches another, we have found an alternating path. This method works fine for bipartite graphs, but on nonbipartite graphs there is a subtle difficulty: a vertex can appear on an alternating path in either parity, where we call a vertex *even*

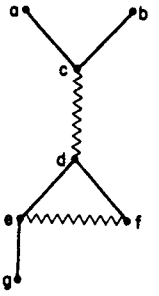


FIG. 9.2. *A graph in which it is hard to find an augmenting path. If we search from a and allow one visit per vertex, labeling c and e odd prevents discovery of the augmenting path* [*a, c, d, f, e, g*]. *Allowing two visits per vertex may produce the supposed augmenting path* [*a, c, d, e, f, d, c, b*].

if it is an even distance from the starting free vertex and *odd* otherwise. (Edmonds [3] called even vertices "outer" and odd vertices "inner".) If during the search we do not allow two visits to a vertex, one in each parity, we may miss an augmenting path; if we allow visits in both parities we may generate a supposedly augmenting path that is not simple. (See Fig. 9.2.)

Such an anomaly can only occur if G contains the configuration shown in Fig. 9.3, consisting of an alternating path p from a free vertex s to an even vertex v and an edge from v to another even vertex w on p. We call the odd-length cycle formed by $\{v, w\}$ and the part of p from w to v a *blossom*; vertex w is the *base* of the blossom and the part of p from s to w is the *stem* of the blossom. Edmonds [3] discovered how to

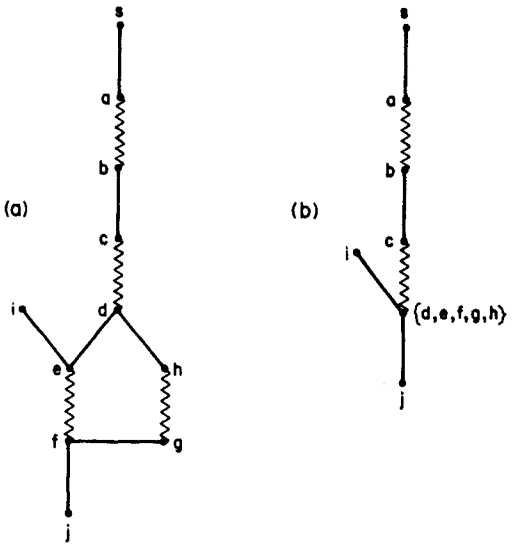


FIG. 9.3. *Shrinking a blossom.* (a) *Blossom defined by path from s to d and cycle* [*d, e, f, g, h, d*]. *Vertex d is the base.* (b) *Shrunken blossom. Augmenting path from s to i*(*j*) *corresponds to augmenting path in original graph going around blossom clockwise (counterclockwise).*

deal with this situation: We shrink the blossom to a single vertex, called a *shrunken blossom,* and look for an augmenting path in the shrunken graph G.

In our discussion we shall sometimes not distinguish between the expanded and shrunken forms of a blossom; the graph being considered will resolve this ambiguity. The following theorem justifies blossom-shrinking:

THEOREM 9.3. *If G' is formed from G by shrinking a blossom b, then G' contains an augmenting path if and only if G does.*

Proof (only if). Suppose G' contains an augmenting path p. If p avoids b, then p is an augmenting path in G. If p contains b, either b is a free vertex or p contains the matching edge incident to b. In either case expansion of the blossom either leaves p an augmenting path or breaks p into two parts, one of which contains the base of blossom, that can be reconnected to form an augmenting path by inserting a path going around the blossom in the appropriate direction from the base (see Fig. 9.3). Thus G contains an augmenting path. □

The "if" direction of Theorem 9.3 is harder to prove; we shall obtain it by proving the correctness of an algorithm developed by Edmonds [3] that finds augmenting paths using blossom-shrinking. The algorithm consists of an exploration of the graph that shrinks blossoms as they are encountered. The algorithm builds a forest consisting of trees of alternating paths rooted at the free vertices. For purposes of the algorithm we replace every undirected edge $\{v, w\}$ by a pair of directed edges $[v, w]$ and $[w, v]$. Each vertex is in one of three states: *unreached, odd,* or *even.* For any matched vertex v, we denote by *mate* (v) the vertex w such that $\{v, w\}$ is a matching edge. For each vertex v the algorithm computes $p(v)$, the parent of v in the forest. Initially every matched vertex is unreached and every free vertex v is even, with $p(v) =$ **null.** The algorithm consists of repeating the following step until an augmenting path is found or there is no unexamined edge $[v, w]$ with v even (see Fig. 9.4):

EXAMINE EDGE (Edmonds). Choose an unexamined edge $[v, w]$ with v even and examine it, applying the appropriate case below:

Case 1. w is odd. Do nothing. This case occurs whenever $\{v, w\}$ is a matching edge and can also occur if $\{v, w\}$ is free.

Case 2. w is unreached and matched. Make w odd and *mate* (w) even; define $p(w) = v$ and $p(\textit{mate}\ (w)) = w$.

Case 3. w is even and v and w are in different trees. Stop; there is an augmenting path from the root of the tree containing v to the root of the tree containing w.

Case 4. w is even and v and w are in the same tree. Edge $\{v, w\}$ forms a blossom. Let u be the nearest common ancestor of v and w. Condense every vertex that is a descendant of u and an ancestor of v or w into a blossom b; define $p(b) = p(u)$ and $p(x) = b$ for each vertex x such that $p(x)$ is condensed into b.

We call this the *blossom-shrinking algorithm.* A vertex (either an original or a blossom) is *shrunken* if it has been condensed into a blossom (and thus no longer appears in the graph); any odd, even, or shrunken vertex is *reached.* We regard a blossom b as containing not only the vertices on the cycle forming b but also all

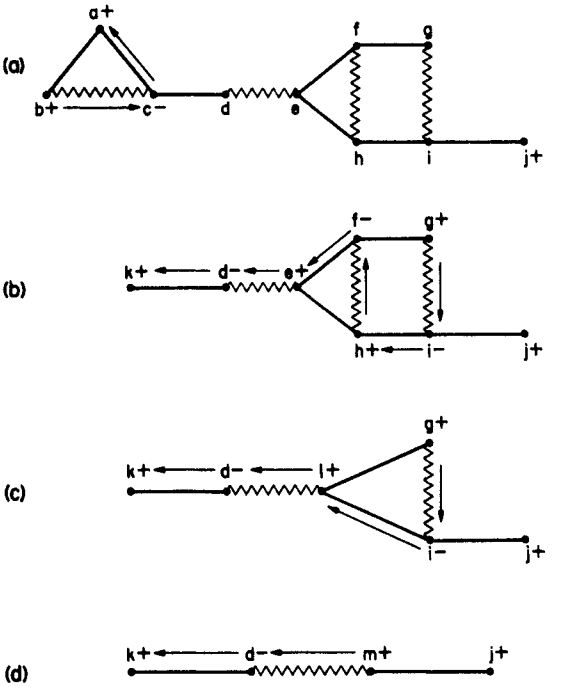


FIG. 9.4. *Execution of the blossom-shrinking algorithm. Plus denotes an even vertex, minus an odd vertex. Arrows denote parents.* (a) *After examining* $[a, c]$ (*Case* 2). (b) *After examining* $[b, a]$ (*Case* 4), $[c, d]$, $[e, f]$, $[h, i]$, *and* $[g, f]$ (*Case* 1). *Vertex* $k = \{a, b, c\}$. (c) *After examining* $[e, h]$. *Vertex* $l = \{e, f, h\}$. (d) *After examining* $[l, g]$. *Vertex* $m = \{g, i, l\}$. *On examining* $[j, m]$ (*Case* 3), *the algorithm halts with success.*

shrunken vertices combined through repeated shrinking to form the blossoms on the cycle; that is, we treat containment as transitive.

THEOREM 9.4. *The blossom-shrinking algorithm succeeds* (*stops in Case* 3) *if and only if there is an augmenting path in the original graph.*

Proof. If the algorithm succeeds, there is an augmenting path in the current shrunken graph. This path can be expanded to an augmenting path in the original graph by expanding blossoms in the reverse order of their shrinking, reconnecting the broken parts of the path each time a blossom on the path is expanded, as described in the proof of Theorem 9.3 (only if).

To prove the converse, we first note several properties of the algorithm. If v is a reached, matched vertex, then *mate* (v) is also reached. If v is a shrunken, free vertex, then v is contained in a free blossom. If the algorithm stops with failure, any two even or shrunken vertices that were adjacent at some time during the computation are condensed into a single blossom when the algorithm halts. To verify this third claim, suppose to the contrary that $\{v, w\}$ is an edge such that v and w are both even or shrunken. Without loss of generality suppose v became even or shrunken after w. Eventually either v and w will be condensed into a common

blossom, or an edge corresponding to $[v, w]$ in the current shrunken graph will be examined; such an examination causes v and w to be condensed into a common blossom.

Suppose the algorithm fails but there is an augmenting path $p = [x_0, x_1, \cdots, x_{2l+1}]$. Consider the situation after the algorithm halts.

It suffices to show that x_i is even (or shrunken) for all even i, $0 \leqq i \leqq 2l$. For then symmetry implies that x_i is even (or shrunken) for all i, $0 \leqq i \leqq 2l + 1$, a contradiction. Thus let i be the least even index such that x_i is not even or shrunken. Observe that x_{i-1} is not even: $i > 0$ and x_{i-1} is the mate of x_i. Further x_{i-1} is not odd. Since x_{i-2} is even, x_{i-1} is reached, which implies x_{i-1} is even. Let j be the smallest index less than $i-1$ such that $x_j, x_{j+1}, \cdots, x_{i-1}$ are even. All of these vertices are in the same blossom. But this blossom has two bases: x_{i-1} is the base, since its mate x_i is not in the blossom; x_j is the base, since its mate is not in the blossom (j is even, and either $j > 0$ and its mate is x_{j-1}, or $j = 0$ and it has no mate). This is impossible, which implies that the algorithm must halt with success if there is an augmenting path. □

Theorem 9.4 implies the "if" part of Theorem 9.3. Let G' be formed from G by shrinking a blossom b. Suppose we run the algorithm in parallel on G and G'. On G, we begin by following the path to and around the blossom and shrinking it. On G', we begin by following the path to b. Now the algorithm is in exactly the same state on G and G', and it will succeed on G if and only if it succeeds on G'.

We conclude this section with two easy-to-prove observations about the blossom-shrinking algorithm and its use in the augmenting path method. After performing an augmentation, we need not immediately expand all blossoms; expansion is required only when a blossom is on an augmenting path or when it becomes an odd vertex. Suppose that while searching for an augmenting path we generate a tree such that every edge $[v, w]$ with v in the tree has been examined and every edge $[w, v]$ with v but not w in the tree has v odd. (Edmonds called such a tree *Hungarian*.) Then we can permanently delete from the graph all vertices in the tree and in its blossoms; none will ever again be on an augmenting path, no matter what augmentations occur.

9.4. Algorithms for nonbipartite matching. The augmenting path method, using blossom-shrinking to find augmenting paths, will find a maximum size matching in polynomial time. Edmonds claimed an $O(n^4)$ time bound, which is easy to obtain; see the book of Papadimitriou and Steiglitz [21]. Witzgall and Zahn [22] gave a related algorithm that instead of shrinking blossoms uses a vertex labeling scheme to keep track of the blossoms implicitly; they did not discuss running time. Balinski [1] gave a similar algorithm that runs in $O(n^3)$ time. Both Gabow [9], [10] and Lawler [18] discovered how to implement Edmonds's algorithm to run in $O(n^3)$ time. As Gabow noted, the running time can be reduced to $O(nm\alpha(m, n))$ using the disjoint set union algorithm discussed in Chapter 2. The linear-time set union algorithm of Gabow and Tarjan [11] further reduces the running time, to $O(nm)$. We shall describe how to implement blossom-shrinking to attain the best of these bounds.

The hard part is to keep track of blossoms. We do this by manipulating only the

vertices in the original graph. Each vertex v in the current shrunken graph corresponds to the set of original vertices condensed to form it. At any time during the running of the algorithm these sets partition the original vertices; we maintain this partition using the operations makeset, link, and find defined in Chapter 2. We define the *origin* of a vertex v in the shrunken graph inductively to be v if v is an original vertex or the origin of the base of v if v is a blossom. We label the canonical vertex of each set (see Chapter 2) with the origin of the current vertex corresponding to the set; that is, if v is an original vertex, *origin* (find (v)) is the origin of the current vertex into which v has been condensed.

We represent the vertices in the current shrunken graph by their origins. Instead of modifying the edges of the graph as blossoms are shrunk, we retain the original edges and use origin and find to convert them into edges in the shrunken graph. More specifically, let $v' = origin$ (find (v)) for any original vertex v. Then $[v', w']$ is the current edge corresponding to original edge $[v, w]$. Note that if v is unreached or odd, $v' = v$.

As we explore the graph we generate a spanning forest, which we represent by defining *predecessors* of the odd vertices. When examination of an original edge $[v, w]$ causes an unreached vertex w to become odd, we define *pred* (w) = v. From predecessors and mates we can compute parents in the forest as follows: if v is an origin, its parent $p(v)$ is *mate* (v) if v is even, *pred* (v)' if v is odd; we assume that *mate* (v) = **null** if v is a free vertex.

We also compute certain information necessary to construct an augmenting path. For each odd vertex v condensed into a cycle we define a *bridge*. Suppose the examination of an original edge $[v, w]$ causes a blossom to form containing odd vertex x. We define *bridge* (x) to be $[v, w]$ if x is an ancestor of v' before condensing or to be $[w, v]$ if x is an ancestor of w'.

Initialization for each vertex v consists of defining *origin* (v) = v, executing makeset (v), and making v even if it is free and unreached if it is matched. To execute the algorithm we repeat the following step until detecting an augmenting path or running out of unexamined edges $[v, w]$ such that v' is even (see Fig. 9.5):

EXAMINE EDGE. Choose an unexamined edge $[v, w]$ such that v' is even and examine it, applying the appropriate case below.

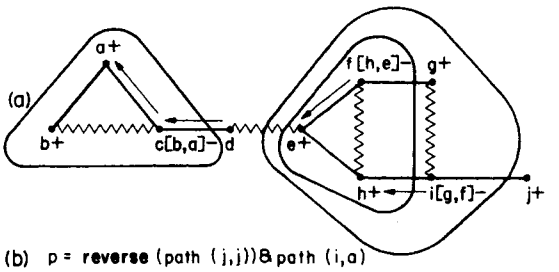
Case 1. w' is odd. Do nothing.

Case 2. w' is unreached. Make w' odd and *mate* (w') even; define *pred* (w') = v.

Case 3. w' is even and v' and w' are in different trees. Stop; there is an augmenting path.

Case 4. w' is even, $v' \neq w'$, and v' and w' are in the same tree. A blossom has been formed. Let u be the nearest common ancestor of v' and w'. For every vertex x that is a descendant of u and an ancestor of v', perform link (find (u), find (x)) and if x is odd define *bridge* (x) = $[v, w]$. For every vertex x that is a descendant of u and an ancestor of w', perform link (find (u), find (x)) and if x is odd define *bridge* (x) = $[w, v]$. Define *origin* (find (u)) = u.

To complete the implementation we must fill in a few more details. We need a way to choose unexamined edges $[v, w]$ such that v' is even. For this purpose we



(b) p = **reverse** (path (j,j)) & path (i,a)

= [j] & [i] & **reverse** (path (g,g)) & path (f,a)

= [j, i, g] & [f] & **reverse** (path (h,h)) & path (e,a)

= [j, i, g, f, h] & [e, d] & path (c,a)

= [j, i, g, f, h, e, d] & [c] & **reverse** (path (b,b)) & path (a,a)

= [j, i, g, f, h, e, d, c, b, a].

FIG. 9.5. *Efficient implementation of blossom-shrinking.* (a) *Labeling of graph in Fig. 9.4. Plus denotes an even, minus an odd vertex; arrows denote predecessors. Edges next to shrunken odd vertices are bridges. Blossoms are circled. The origins of k, l, m are a, e, e, respectively.* (b) *Construction of augmenting path.*

maintain the set of such edges, from which we delete one edge at a time. Initially the set contains all edges $[v, w]$ such that v is free. When an unreached vertex v becomes even or an odd vertex v is condensed into a blossom, we add all edges $[v, w]$ to the set. By varying the examination order we can implement various search strategies.

We also need a way to distinguish between Case 3 and Case 4 and to determine the set of edges to be condensed into a blossom if Case 4 applies. When examining an edge $[v, w]$ such that w' is even, we ascend through the forest simultaneously from v' and from w', computing $v_0 = v'$, $w_0 = w'$, v_1, w_1, v_2, w_2, $\cdots$, where $v_{i+1} = p(v_i)$ and $w_{i+1} = p(w_i)$. We stop when reaching different free vertices from v and from w (Case 3 applies), or when reaching from w' a vertex u previously reached from v' or vice versa (Case 4 applies). In the latter case u is the nearest common ancestor of v' and w', and the blossom consists of the vertices $v_0, v_1, \cdots, v_j = u$ and $w_0, w_1, \cdots, w_k = u$. The number of vertices visited by this process is $O(n)$ in Case 3, at most twice the number of vertices on the cycle defining the blossom in Case 4.

The total number of vertices on all blossom cycles is at most $2n - 2$, since there are at most $n - 1$ blossoms and shrinking a blossom of k vertices reduces the number of vertices in the graph by $k - 1$. A simple analysis shows that the disjoint set operations, of which there are n makeset, at most $n - 1$ link and $O(m)$ find operations, dominate the running time of the algorithm. If we use the data structure of Chapter 2 to implement makeset, link and find, the time to detect an augmenting

path is $O(m\alpha(m, n))$. The Gabow–Tarjan set union algorithm [11] is also usable and reduces the running time to $O(m)$.

There remains the problem of constructing an augmenting path once one is detected. Suppose the algorithm stops in Case 3, having found an edge $[v, w]$ such that v' and w' are even and in different trees. Let x be the root of the tree containing v' and y the root of the tree containing w'; the algorithm determines x and y in the process of detecting an augmenting path. Then **reverse** (*path* (v, x)) & *path* (w, y) is an augmenting path, where **reverse** reverses a list (see Chapter 1) and *path* is defined recursively as follows (see Fig. 9.5):

$$path\ (v, w) = \begin{cases} [v] \text{ if } v = w, \\ [v, mate\ (v)]\ \&\ path\ (pred\ (mate\ (v)), w) \text{ if} \\ \quad v \neq w \text{ and } v \text{ is even}, \\ [v]\ \&\ \textbf{reverse}\ (\ path\ (x, mate\ (v)))\ \&\ path\ (y, w) \text{ if} \\ \quad v \neq w \text{ and } v \text{ is odd, where } [x, y] = bridge\ (v). \end{cases}$$

The function *path* (v, w) defines an even-length alternating path from v to w beginning with a matching edge, under the assumption that at some time during the running of the blossom-shrinking algorithm v' is a descendant of w in the forest. An induction on the number of blossoms shrunk verifies that *path* is correct. The time required to compute *path* (v, w) is proportional to the length of the list returned, since with an appropriate implementation of lists we can perform concatenation and reversal in $O(1)$ time (see §1.3). With this method the time needed to construct an augmenting path is $O(n)$, and the time to find a maximum size matching is either $O(nm\alpha(m, n))$ or $O(nm)$ depending on the disjoint set implementation.

This algorithm is not the last word on unweighted nonbipartite matching. Even and Kariv [6], [14], in a remarkable tour-de-force, managed to generalize the Hopcroft–Karp bipartite matching algorithm by including blossom-shrinking. Their algorithm, though complicated, runs in $O(\min\{n^{2.5}, \sqrt{n}\, m \log \log n\})$ time. Micali and Vazirani [19], using the same ideas, obtained a simplified algorithm with a running time of $O(\sqrt{n}\, m)$. Thus the best time bounds for unweighted bipartite and nonbipartite matching are the same.

The situation is similar for weighted nonbipartite matching. Edmonds [4] obtained an $O(n^4)$-time algorithm that combines maximum weight augmentation (Theorem 9.2) with blossom-shrinking. With Edmonds's method it is necessary to perserve shrunken blossoms from augmentation to augmentation, only expanding or shrinking a blossom under certain conditions determined by the search for a maximum weight augmenting path. Gabow [9] and Lawler [18] independently discovered how to implement this algorithm to run in $O(n^3)$ time. Recently Galil, Micali, and Gabow [12] reduced the time bound to $O(n\, m \log n)$ by using an extension of meldable heaps (see Chapter 3) to speed up the search for an augmenting path. Thus the best known time bound for weighted nonbipartite matching is $O(\min\{n^3, n\, m \log n\})$. This is slightly larger than the best known bound for the bipartite case, $O(n\, m \log_{(2+m/n)} n)$. Curiously, the bound *is* the same as the

best known bound for maximum network flow, although the algorithms for the two problems use different data structures and techniques. We conjecture that there is an $O(n\,m \log (n^2/m))$-time algorithm for weighted bipartite matching.

References

[1] M. L. BALINSKI, *Labelling to obtain a maximum matching*, in Proc. Combinatorial Mathematics and its Applications, North Carolina Press, Chapel Hill, NC, 1967, pp. 585–602.

[2] C. BERGE, *Two theorems in graph theory*, Proc. Natl. Acad. Sci., 43 (1957), pp. 842–844.

[3] J. EDMONDS, *Paths, trees, and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[4] ———, *Matching and a polyhedron with* 0-1 *vertices*, J. Res. Nat. Bur. Standards Sect. B, 69 (1965), pp. 125–130.

[5] J. EGERVÁRY, *Matrixok kombinatórius tulajdonságairól*, Mat. Fiz. Lapok, 38 (1931), pp. 16–28. (In Hungarian.)

[6] S. EVEN AND O. KARIV, *An $O(n^{2.5})$ algorithm for maximum matching in general graphs*, in Proc. 16th Annual IEEE Symposium on Foundations of Computer Science, 1975, pp. 100–112.

[7] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.

[8] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[9] H. N. GABOW, *Implementation of algorithms for maximum matching on nonbipartite graphs*, Ph.D. thesis, Dept. Computer Science, Stanford Univ., Stanford, CA, 1973.

[10] ———, *An efficient implementation of Edmonds' algorithm for maximum matching on graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 221–234.

[11] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, 246–251.

[12] Z. GALIL, S. MICALI AND H. GABOW, *Maximal weighted matching on general graphs*, in Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science, 1982, pp. 255–261.

[13] J. E. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matching in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

[14] O. KARIV, *An $O(n^{2.5})$ algorithm for maximum matching in general graphs*, Ph.D. thesis, Dept. Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 1976.

[15] D. KÖNIG, *Graphen und Matrizen*, Mat. Fiz. Lapok, 38 (1931), pp. 116–119.

[16] ———, *Theorie der endlichen und unendlichen Graphen*, Chelsea, New York, 1950.

[17] H. W. KUHN, *The Hungarian method for the assignment problem*, Naval Res. Logist. Quart., 2 (1955), pp. 83–98.

[18] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[19] S. MICALI AND V. V. VAZIRANI, *An $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding maximum matching in general graphs*, in Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, 1980, pp. 17–27.

[20] R. Z. NORMAN AND M. O. RABIN, *An algorithm for a minimum cover of a graph*, Proc. Amer. Math. Soc., 10 (1959), pp. 315–319.

[21] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Networks and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[22] C. WITZGALL AND C. T. ZAHN, JR., *Modification of Edmonds' maximum matching algorithm*, J. Res. Nat. Bur. Standards Sect. B, 69 (1965), pp. 91–98.