Unique Maximum Matching Algorithms*

Harold N. Gabow† Haim Kaplan‡ Robert E. Tarjan§

May 18, 2002

Abstract

We consider the problem of testing the uniqueness of maximum matchings, both in the unweighted and in the weighted case. For the unweighted case, we have two results. First, given a graph with n vertices and m edges, we can test whether the graph has a unique perfect matching, and find it if it exists, in $O(m \log^4 n)$ time. This algorithm uses a recent dynamic connectivity algorithm and an old result of Kotzig characterizing unique perfect matchings in terms of bridges. For the special case of planar graphs, we improve the algorithm to run in $O(n \log n)$ time. Second, given one perfect matching, we can test for the existence of another in linear time. This algorithm is a modification of Edmonds' blossom-shrinking algorithm implemented using depth-first search. A generalization of Kotzig's theorem proved by Jackson and Whitty allows us to give a modification of the first algorithm that tests whether a given graph has a unique f-factor, and find it if it exists. We also show how to modify the second algorithm to check whether a given f-factor is unique. Both extensions have the same time bounds as their perfect matching counterparts. For the weighted case, we can test in linear time whether a maximum-weight matching is unique, given the output from Edmonds' algorithm for computing such a matching. The method is an extension of our algorithm for the unweighted case.

*A preliminary version of part of our work was presented at the 31^{st} Annual ACM Symposium on Theory of Computing [11].

†Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado 80309-0430, USA. e-mail: hal@cs.colorado.edu

‡School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. e-mail: haimk@math.tau.ac.il

§Department of Computer Science, Princeton University, Princeton, NJ 08544 USA and InterTrust Technologies, Sunnyvale, CA. Research at Princeton University partially supported by the NSF, Grant No. CCR-9626862. e-mail: ret@cs.princeton.edu.

Running Head: Unique Maximum Matching Algorithms.

Corresponding author:

Haim Kaplan

Department of Computer Science

School of Mathematics

Tel Aviv University

Tel Aviv, 69978

Israel

Tel: +972-3-640-8814

Fax: +972-3-640-9357

email: haimk@math.tau.ac.il

Final version will be provided in LaTeX.

1 Introduction

Let G be an undirected graph with n vertices and m edges. A *matching* in G is a set of edges, no two incident to the same vertex. In this paper we consider the problem of testing whether G has a unique maximum matching, either in the unweighted or the weighted case.

The unweighted case was considered by Golumbic, et al. [17]. They posed the question, given a matching M, of whether G has another matching on the same set of vertices. They proposed a simple $O(nm)$-time algorithm for this problem. Since we can restrict our attention to the subgraph of G induced by the vertices incident to the edges of M, the problem is equivalent to asking whether a given perfect matching is unique. (A *perfect matching* is a matching meeting every vertex.)

For the unweighted case, we have two results. In Section 2 we describe an $O(m \log^4 n)$-time algorithm to test whether a given graph has a unique perfect matching, and to find one if it exists. This algorithm relies for its correctness on an old theorem of Kotzig [21] characterizing perfect matchings in terms of bridges (edges whose removal increases the number of connected components). It relies for its efficiency on a recent dynamic 2-edge connectivity algorithm [18]. An improvement in dynamic testing for 2-edge components under edge deletion would likely give an improvement in our algorithm. Dynamic 2-edge connectivity testing has also been used recently in an efficient algorithm for finding perfect matchings in 3-regular bridgeless graphs[1] [4]. For the special case of planar graphs, our algorithm can be simplified and its running time improved to $O(n \log n)$ using a decremental 2-edge connectivity algorithm for planar graphs [16]. We describe this result in Section 2.4.

Also, given one perfect matching M, we show how to test whether there is another, and find one if it exists, in $O(m)$ time, improving the result of Golumbic et al. by a factor of n. This problem is equivalent to finding an alternating cycle in the graph (See e.g. [17].) (An *alternating cycle* is a simple cycle alternating edges in M, called *matched edges*, with edges not in M, called *unmatched edges*.) This algorithm combines Edmonds' classical blossom-shrinking algorithm for finding a maximum-cardinality matching [7] with depth-first search. Combining this algorithm with a method of Itai, Rodeh, and Tanimoto [19] gives an efficient algorithm for listing all the perfect (or maximum cardinality) matchings of an arbitrary graph, in $O(m\sqrt{n})$ time for the first matching and $O(m)$ time for each subsequent matching. These results appear in Section 3.

[1] Every such graph is known to have a perfect matching by Peterson's theorem.

Let $G = (V, E)$ be an undirected graph and let f be a function mapping each vertex v to an integer $f(v)$. An *f-factor* in G is a set of edges such that exactly $f(v)$ of them are incident to each vertex v. Jackson and Whitty [20] generalize Kotzig's theorem and characterize unique f-factors in terms of bridges. Specifically, they show that a graph with a unique f-factor either has a vertex for which $f(v) \in \{d(v), 0\}$, where $d(v)$ is the degree of v, or it has a bridge. This theorem allows us to give a modification of our first algorithm, described in Section 4, that tests whether a given graph has a unique f-factor. The running time remains $O(m \log^4 n)$. Jackson and Whitty [20] prove the generalization of Kotzig's theorem by reducing the unique f-factor problem to a unique perfect matching problem on a corresponding graph. This reduction, together with a technique of Gabow [10] for sparsifing the resulting graph, allows us to adapt our second algorithm to test whether a given f-factor is unique, or equivalently find an alternating cycle, in linear time. This algorithm, combined with a straightforward extension of the result of Itai, Rodeh, and Tanimoto to f-factors, allows us to list all f-factors of a given graph, in $O(m\sqrt{F})$ time for the first f-factor and $O(m)$ time for each subsequent f-factor. Here $F = \sum\{f(v) \mid v \in V\}$. These algorithms also appear in Section 4.

For the weighted case, we consider the following problem. Suppose every edge e of G has a real-valued *weight* $w(e)$. A *maximum-weight matching* is a matching whose total weight is maximum. Edmonds' well-known algorithm for finding a maximum-weight matching [6] produces not only such a matching but also additional information that allows verification that the solution is correct. In Section 5 we show how, given the output of Edmonds' algorithm, to test whether the maximum-weight matching is unique in $O(m)$ time. The method is an application of the algorithm in Section 3.

Our algorithm of Section 5 has an application to computational biology. References [5] and [26] describe an automated system that successfully predicts RNA structure. The system is based on a model in which the base pairs in an RNA molecule correspond to the edges of a maximum-weight matching in a certain graph. One expects this maximum-weight matching to be unique, unless the molecule can exhibit more than one folding structure. Our algorithm can be used to verify uniqueness and hence give additional credibility to the model; currently, the system assumes uniqueness.

2 Unique Perfect Matchings and Bridges

Our algorithm for finding a unique perfect matching given a graph G relies on the intimate connection between such matchings and bridges. Let us call a connected component *even* if it contains an even number of vertices and *odd* otherwise. We call a bridge *even* if its removal produces two new even components and *odd* if its removal produces two new odd components. Suppose G has a unique perfect matching M. We observe that, by parity, every component of G is even, every odd bridge of G is in M, and every even bridge of G is not in M. The usefulness to us of this observation comes from the following converse:

Lemma 2.1 (Kotzig [21]; see also [9, 23]) *A unique perfect matching contains a bridge.*

Applied recursively, the observation and Lemma 2.1 give the following algorithm for finding a unique perfect matching. Initialize $M = \emptyset$ and begin with G as the current graph. Repeat the following step until the current graph contains an odd component or does not contain an odd bridge: Find an odd bridge $\{x, y\}$ in the graph, add it to M, and delete x and y and all incident edges. Then G has a unique perfect matching, namely M, if and only if the final graph is empty.

2.1 Refinement of the algorithm

In order to make this algorithm efficient, we must localize the search for odd bridges. The key observation is that every bridge newly created by deletion of an edge $\{v, w\}$ is on every path from v to w. This leads to the following refinement of the algorithm, in which R is the set of unprocessed bridges in the current graph. We call a connected component formed by deletion of all bridges a *2-edge component.* The 2-edge components are vertex-disjoint. Hence there are at most n of them.

Initialize $M = \emptyset$ and R to be the set of all bridges of G.

While $R \neq \emptyset$ repeat the following steps:

  Delete an edge $\{x, y\}$ from R.

  If $\{x, y\}$ is an odd bridge, delete $\{x, y\}$ from G, add $\{x, y\}$ to M, and

   repeat the following steps for each edge $\{v, w\}$ incident to x or y:

    Delete $\{v, w\}$ from G, and from R if it is in R

    If v and w are still connected but are in different 2-edge components, then find a path $P(v, w)$ connecting v and w and add every bridge on $P(v, w)$ to R.

Lemma 2.2 *When the refined algorithm is run, the final graph contains no odd bridges.*

Proof. Once an edge becomes a bridge, it remains a bridge until it is deleted, because edge deletion preserves bridges. Furthermore, once an odd component is created outside the inner loop, it remains intact until the algorithm halts. This means that an odd bridge, once created, remains an odd bridge unless the component containing it becomes odd. Thus it suffices to show that if an edge ever becomes a bridge it is added to R. Suppose deletion of an edge $\{v, w\}$ creates a new bridge $\{r, s\}$. Then deletion of $\{r, s\}$ must separate v and w, which means that $\{r, s\}$ is on every path joining v and w, and v and w must be in different 2-edge components (after deletion of $\{v, w\}$). Hence $\{r, s\}$ is added to R. Observe also that such an edge $\{r, s\}$ is not a bridge before deletion of $\{v, w\}$, because $P(v, w)$ and $\{v, w\}$ form a cycle. This means that an edge is added to R at most once. $\Box$

We conclude immediately from Lemma 2.2 that the refined algorithm is correct.

2.2 Implementation and Efficiency

Simple list data structures suffice to implement all of the algorithm in linear time (each edge is added to R at most once) except for testing if $\{x, y\}$ is an odd bridge and the last statement in the inner loop. We implement these steps using a data structure of [18]. The dynamic 2-edge component algorithm described there supports the following operations: $insert(v, w)$, which adds edge $\{v, w\}$ to the graph; $delete(v, w)$, which deletes edge $\{v, w\}$ from the graph; 2-$conn(v, w)$, which returns true exactly when v and w are in the same 2-edge component of the graph; $conn(v, w)$, which returns true exactly when v and w are in the same connected component of the graph; and $comp\_size(v)$, which returns the number of vertices in the component containing v. (The last two operations are implemented in the dynamic connected component algorithm of [18], although they are also easy to implement in the 2-edge component algorithm.) The time for each of these operations is $O(\log^4 n)$. This is an amortized bound for the first two operations and worst-case for the others.

We test if $\{x, y\}$ is an odd bridge using the operations *delete*, *comp_size* and *insert*. We test if v and w are connected and in different 2-edge components using *conn* and 2-*conn*. It remains only to find a path $P(v, w)$ and to find the bridges on $P(v, w)$. This is done as follows. We assume some familiarity with [18].

The data structure of [18] maintains a spanning forest of G. Each tree T in this spanning forest is represented by a *top-tree* $\mathcal{T}$ [1]. The top tree is a data structure used to represent trees that are subject to insertion and deletion of edges. The top tree structure is a variant of Frederickson's topology trees [8] that directly supports trees of unbounded degree. Top trees are also related to the dynamic trees of Sleator and Tarjan [25]. Top trees have been used to help solve several dynamic connectivity problems as well as to maintain centers, medians, and diameters of trees that are subject to insertion and deletion of edges [1, 2].

A top tree $\mathcal{T}$ is a binary tree representing an arbitrary tree T along with a set of at most two nodes of T called *external boundary nodes*. A top tree has the following structure.

1. Each node of a top-tree represents a *cluster*. A *cluster* is a subtree that has one or two *boundary nodes*, where a *boundary node* is either an external boundary node or a node adjacent to an edge outside the subtree.

2. Each leaf of $\mathcal{T}$ represents an edge of T. The boundary nodes of the leaf are the endpoints of the edge.

3. The root of $\mathcal{T}$ represents T itself.

4. Let C be the cluster represented by an internal node x of $\mathcal{T}$, and let A and B be the clusters represented by the children of x. Then A and B share a single node (which is also a common boundary node). Furthermore C is the tree formed by combining A and B.

5. The depth of $\mathcal{T}$ is $O(\log n)$, where n is the number of nodes of T.

If vertices x and y are in T, let $P(x, y)$ denote the (unique) path in T from x to y. A cluster with two distinct boundary nodes a and b is called a *path cluster*, and we say that $P(a, b)$ is its *cluster path.*

Top trees support the operation $expose(v, w)$. This operation modifies the top tree $\mathcal{T}$ so that its external boundary nodes become v and w. By the definition above, v and w then become the boundary nodes of the root cluster. The cluster path of the root cluster, $P(v, w)$, will be used as the path $P(v, w)$ of our algorithm. The path $P(v, w)$ is not constructed explicitly but is maintained implicitly by the data structure.

To find a path $P(v, w)$ and find the bridges on $P(v, w)$, we begin by performing $expose(v, w)$. Then we traverse one or more paths in $\mathcal{T}$, each one going from the root of $\mathcal{T}$ to a leaf. These leaves will be precisely the bridges on $P(v, w)$. (Recall that the leaf clusters are the edges of T.) The traversal will visit precisely the nodes of $\mathcal{T}$ whose clusters contain bridges on $P(v, w)$. The traversal works as follows.

Suppose we are visiting an internal node C of $\mathcal{T}$. C has two children. We will visit one or both children of C, following the rule that we visit a child whose cluster contains at least one of our desired bridges. In more detail, C will be a path-cluster, say with boundary nodes a, c. If only one child of C is a path-cluster then visit it. Suppose both children are path-clusters. Denote the children by $C_{a,b}$ and $C_{b,c}$, where $C_{a,b}$ ($C_{b,c}$) has boundary nodes a, b (b, c). By definition, b is a vertex on $P(a, c)$. Visit $C_{a,b}$ if a and b are in different 2-edge components. Process $C_{b,c}$ similarly.

To prove that this traversal is correct it suffices to show that we visit $C_{a,b}$ if and only if $P(a, b)$ contains a bridge. This follows from the fact that two vertices joined by a simple path P are separated by a bridge on P exactly when they are in different 2-edge components.

It remains to indicate how we check that a and b are in different 2-edge components. This is easy to do because in the top trees of [18] each cluster C in $\mathcal{T}$ stores a value c_C that equals -1 precisely when the boundary nodes of the cluster are in different 2-edge components. Hence we need only read the c_C value of the child $C_{a,b}$. In truth the algorithm stores c_C values as lazy information; whenever we traverse an edge from parent to child, we must update the c_C value of the child using update information stored in the parent. Our traversal can do this, so the desired information is available for our test.

Consider the time spent by the above traversal to discover bridges. Each bridge is discovered by traversing a path from the root to a leaf of $\mathcal{T}$. Such a path contains $O(\log n)$ nodes. We spend $O(1)$ time at each node (including time to propagate c_C values). Hence each bridge is discovered in $O(\log n)$ time.

The algorithm discovers at most n bridges. This follows since each bridge discovered either gets deleted, thereby increasing the number of 2-edge components, or is a bridge of the final graph. Furthermore, as noted in the proof of Lemma 2.2, each bridge is discovered only once. Hence the total time for discovering bridges is $O(n \log n)$.

The rest of the algorithm uses time $O(m \log^4 n)$ time by the complexity bound of [18]. This gives an $O(m \log^4 n)$ time bound for the entire algorithm. The space required is $O(m)$.

2.3 Dense Graphs

For the case of dense graphs, we can improve the running time by a polylogarithmic factor. We note first that Thorup [30] has proposed a Las Vegas randomized algorithm for decremental 2-edge connectivity that has an expected running time of $O(m \log(n^2/m) + n(\log n)^6 (\log \log n)^2)$. Note that this bound is $O(m)$ if $m = \Theta(n^2)$. This algorithm can be adapted as in Section 2.2 to solve our problem in the same time bound. If we desire a deterministic algorithm, the following alternative approach results in an $O(n^2 + m \log^2 n)$-time algorithm.

We use the naive algorithm given at the beginning of Section 2, which successively deletes odd bridges and the edges incident to their ends. To speed up the computation, we maintain a *sparse certificate* S that contains $O(n)$ edges and has the same bridges and 2-edge components as the current graph G'. Each time we delete a bridge and the edges incident to its ends from G', we first update the sparse certificate S. Then we perform a depth-first search to find new odd bridges of S. Since S contains $O(n)$ edges, this step takes $O(n)$ time [27], and the total time for all such steps is $O(n^2)$.

The remaining detail is the definition and maintainance of S. We maintain S as the union of the edges of F_1 and F_2, where F_1 is a spanning forest of G' and F_2 is a spanning forest of $G' - F_1$. For a proof that the graph induced by S has the same bridges and 2-edge components as G' see [24].

We initialize F_1 to be any spanning forest of G, and we use the dynamic connectivity algorithm of [18], also used in Section 2.2, to maintain the connected components of $G' - F_1$. The spanning forest maintained by this algorithm is F_2. When we delete a bridge and all edges incident to its ends from G', we delete these edges from F_1 and $G' - F_1$, doing appropriate *delete* operations. Now F_1 may no longer be a spanning forest of G', although F_2 has been updated to be a spanning forest of $G' - F_1$. We do a search of $F_1 \cup F_2$ to find edges of F_2 to add to F_1 to restore its being a spanning forest of G'. This takes $O(n)$ time. We move these edges from F_2 to F_1 and do corresponding *delete* operations to update F_2. The total time spent is $O(n^2)$ plus $O(m \log^2 n)$ for the dynamic connectivity operations. Note that the dynamic connectivity maintenance needed here is the "deletion-only" case.

2.4 Planar Graphs

For planar graphs, we can implement a variant of the original (unrefined) unique perfect matching algorithm that runs in $O(n \log n)$ time, using the decremental 2-edge connectivity algorithm for planar graphs of Giammarresi and Italiano [16]. This algorithm runs in a total of $O(n \log n)$ time for n edge deletions. It explicitly maintains the set of bridges as well as the set of 2-edge components. The algorithm maintains the cycles bounding the faces of the graph (an edge is a bridge if and only if it occurs twice on the same cycle), and it obtains its running time by using a "relabel the smaller half" idea, which we also use.

Our unique perfect matching algorithm for planar graphs operates as follows. The algorithm maintains, for each connected component, the parity of its number of vertices. Initializing these values takes $O(n)$ time. If any initial component is odd, the algorithm stops with failure. Otherwise, the algorithm initializes M to be empty and B to be the initial set of bridges, and repeats the following steps until B is empty: Select a bridge $\{v, w\}$ in B. Delete $\{v, w\}$ from the graph. Search the connected components containing v and w concurrently, updating their parities and stopping when one is completely searched. If this component is odd, $\{v, w\}$ is an odd bridge; add it to M and delete all edges incident to v and to w. When deleting an edge, if the deletion creates two new components, concurrently search both components to update the parities, stopping when one is completely searched. When the algorithm finishes, M is a unique perfect matching in the original graph if and only if the final graph is empty.

The total time for searches to update parities is $O(n \log n)$, since we can charge the time for such a search to the smaller of the two new components formed by the bridge deletion, and this component is at most half the size of the component containing the bridge before it was deleted. An easy induction shows that the total charge is $O(n \log n)$. The total time to maintain bridges using the algorithm of [16] is also $O(n \log n)$. Thus the entire algorithm runs in $O(n \log n)$ time. The space required is $O(n)$ [16].

3 Testing for an Alternating Cycle

Suppose now that we are given a perfect matching M in G and asked whether there is another. This is equivalent to testing for the existence of an alternating cycle. Our algorithm for finding an alternating cycle performs a depth-first search [27] of G, traversing alternating paths (paths that alternate matched and unmatched edges). The search continues until it discovers an alternating cycle or runs out of edges to traverse. Like Edmonds' algorithm [7], it occasionally shrinks a blossom (an odd-length simple cycle whose matched edges contain all but one of the vertices on the cycle). It differs from Edmonds' algorithm in five ways:

(1) It defines blossoms slightly differently. (No alternating path to an unmatched vertex is required.)

(2) It starts searching at unexamined matched vertices, rather than unexamined unmatched vertices.

(3) It stops when it finds an alternating cycle, not an augmenting path.

(4) It searches in depth-first order, rather than any order.

(5) It delays blossom-shrinking, rather than shrinking blossoms as soon as they are detected.

Differences (1), (2), and (3) merely reflect the different purpose of our algorithm (detection of an alternating cycle, not an augmenting path). Differences (4) and (5), restricting the search order and delaying blossom-shrinking, are crucial to the correctness of our algorithm. Essentially the same algorithm as ours is used in [13] to find a maximal set of augmenting paths.

We describe the algorithm in Section 3.1. In Section 3.2, we derive critical properties of the algorithm, leading to a proof of correctness. Section 3.3 provides details of the implementation of blossom-shrinking. Section 3.4 describes how to apply the algorithm to the problem of listing all maximum cardinality matching of a given graph. Throughout, we assume familiarity with depth-first search, as described e.g. in [27], and with Edmonds' algorithm [7], as described e.g. in [28].

3.1 The Algorithm

We apply depth-first search to a directed version of the graph. (Every originally undirected edge is directed in both directions.) We represent the matched edges by a function $mate(v)$ on vertices that gives the unique w such that $\{v, w\} \in M$. We represent each unmatched edge $\{v, w\}$ by a pair of directed arcs, (v, w) and its reversal (w, v). During the search, each vertex is labeled *inner* or *outer* as it is reached, and each arc is marked as it is traversed. The algorithm also builds a depth-first spanning forest by assigning a parent to each labeled vertex. Each vertex is *active* for the period of time during which it and its descendants are being labeled. Initially, all vertices are unlabeled and all arcs are unmarked.

The algorithm occasionally shrinks together a set of vertices forming a blossom. The vertices being combined form a path in the spanning forest joining an outer vertex v with one of its descendants w. The shrinking replaces each vertex $x \neq v$ on the path by v. An arc (x, y) or (y, x) with y not on the path is replaced by (v, y) or (y, v), respectively, marked or unmarked according to whether (x, y) or (y, x) is marked or unmarked, respectively. Such replacements can create multiple arcs, which we allow. An arc (x, y) with both x and y on the path is deleted. We call an arc (v, w) a *replacement* of an arc (x, y) if (v, w) is formed from (x, y) by a sequence of one or more replacements. That is, we extend the replacement relation to make it transitive.

Here is the outer loop of the algorithm. The function $p(v)$ gives the parent of vertex v in the spanning forest. If the outer loop runs to completion, the algorithm stops and reports that there are no negative cycles.

While some vertex r is unlabeled, perform the following steps:

label r inner, label $mate(r)$ outer, set $p(r) = null$ and $p(mate(r)) = r$, and perform $scan(mate(r))$.

After $scan(v)$ is performed, we say vertex v is *scanned.* (Every vertex can be labeled but only outer vertices can be scanned.) Recursive procedure *scan* works as follows:

$scan(v)$: make $mate(v)$ and v active. While some arc (v, w) is unmarked, mark it and apply the appropriate case below. Once all arcs (v, w) are marked, make v and $mate(v)$ inactive.

Case 1: w is unlabeled. Label w inner, label $mate(w)$ outer, set $p(w) = v$ and $p(mate(w)) = w$ and recursively perform $scan(mate(w))$.

Case 2: w is inner and active. Stop and report an alternating cycle.

Case 3: w is outer and inactive. (In this case, as we shall see, w is a descendant of v in the spanning forest.) The tree path joining w and v plus the edge $\{v, w\}$ defines a blossom. Shrink this blossom to the single vertex v.

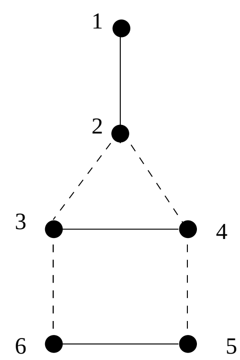

Figure 1: Solid edges are matched, dashed edges unmatched. Immediate shrinking of the blossom with vertices $2, 3, 4$ destroys the alternating cycle $3, 4, 5, 6$ (converting it into a blossom).

Case 4: w is inner and inactive or w is outer and active. Do nothing.

As part of our proof of correctness, we must show that whenever Case 3 occurs, vertex w is indeed a descendant of vertex v. For the moment, let us suppose that the algorithm halts with an error should Case 3 ever occur with w not a descendant of v.

The most subtle point about this algorithm is the delayed blossom-shrinking. If Case 4 occurs with w outer and active, a blossom exists, consisting of the tree path joining v and w plus the edge $\{v, w\}$. (In this case w must be an ancestor of v.) Shrinking such a blossom immediately, which corresponds to what Edmonds' algorithm does, leads to incorrect behavior, as shown by the example in Figure 1. Suppose the search starts with vertex 1, marks arc $(2, 3)$ (Case 1), and then marks $(4, 2)$ (Case 4). Shrinking the blossom $2, 3, 4$ would eliminate the only alternating cycle, $3, 4, 5, 6$. What the algorithm actually does is mark $(4, 5)$ (Case 1) and then mark $(6, 3)$ (Case 2), correctly discovering the alternating cycle.

3.2 Correctness of the algorithm

Because of the delayed blossom-shrinking, proving the correctness of our algorithm is more complicated than proving the correctness of Edmonds' algorithm, which itself is not straightforward. The main burden of the proof is to show that blossom-shrinking preserves both the existence and the non-existence of an alternating cycle. One half (preserving non-existence) is easy; the other half is considerably harder.

We begin with some routine observations about the algorithm. Vertices can be shrunk into a given vertex v only in Case 3: v is the vertex currently being scanned and the vertices shrunk into v are descendants of v. If follows that blossom-shrinking preserves the ancestor-descendant relationship. Furthermore, a vertex is active if and only if it is an ancestor of the vertex currently being scanned. This follows by induction on the number of steps taken by the algorithm; the only interesting case is the blossom-shrinking in Case 3, which preserves the invariant.

Next, we prove a structural lemma about arcs that captures the fact that the search is depth-first and among other things implies that Case 3 never halts with an error.

Lemma 3.1 *If (v, w) is an arc such that v is outer and w is labeled, then either w is a descendant of v or w was labeled before v.*

Proof. Let (v, w) be an arc such that v is outer, w is labeled, and v was labeled before w. We shall show that w was labeled before v is scanned, which implies that w is a descendant of v and hence gives the lemma.

Suppose to the contrary that w is labeled after v is scanned. The arc (v, w) is either an original arc or is created by replacing an arc (x, w) during the execution of *scan*(v). Either (x, w) is marked before its replacement, or (v, w) must be marked during the execution of *scan*(v). In either case w is labeled before v is scanned. □

Corollary 3.2 *If (v, w) is an arc such that v and w are both outer, then v and w are related.*

Proof. Let (v, w) be an arc with both ends outer. Then its reversal (w, v) is such an arc as well. Lemma 3.1 applied to either (v, w) or (w, v) implies that whichever of v and w is labeled first is an ancestor of the other. □

Theorem 3.3 *Case 3 of the algorithm never halts with an error.*

Proof. Let (v, w) be an arc that triggers Case 3. Both v and w are outer. Since w is inactive, w cannot be an ancestor of v. By corollary 3.2, w must be a descendant of v. □

Now we show that if the algorithm halts in Case 2, then the original graph contains an alternating cycle. To verify this, we must show that blossom-shrinking preserves the non-existence of alternating cycles.

Lemma 3.4 *If there is an alternating cycle after a blossom is shrunk, then there was one (possibly a different one) before the shrinking.*

Proof. Consider shrinking a blossom when marking an arc (v,w) in Case 3. Let B be the set of vertices on the blossom, excluding v. Let C be an alternating cycle after the shrinking. Suppose C does not exist before the shrinking. Then C corresponds to a path D in the graph before the shrinking connecting v and some vertex x in B; D contains the edge $\{\text{mate}(v), v\}$ and no vertices in B except x. We can augment D to an alternating cycle by adding the tree path joining v and x if x is outer or the edge $\{v,w\}$ and the tree path joining w and x if x is inner. □

Theorem 3.5 *If the algorithm halts in Case 2, then the original graph contains an alternating cycle.*

Proof. Suppose the algorithm halts by applying Case 2 to an arc (v,w). The edge $\{v,w\}$ and the tree path joining v and w form an alternating cycle in the current, shrunken graph. Using induction to apply Lemma 3.4 to each of the blossom-shrinkings in reverse order, we obtain an alternating cycle in the original graph. □

Finally, we come to the hardest part of the correctness proof, the converse of Theorem 3.5. This result relies on the following lemma about the structure of alternating cycles, whose proof we postpone.

Lemma 3.6 *Let C be an alternating cycle containing a scanned vertex x. Then C contains an outer vertex $w \neq x$ that is an ancestor of x.*

Lemma 3.6 allows us to prove first that blossom-shrinking preserves the existence of alternating cycles and second that the converse of Theorem 3.5 holds.

Lemma 3.7 *If there is an alternating cycle before a blossom is shrunk, then there is one after the shrinking.*

Proof. Consider the shrinking of a blossom formed by an arc (v,w) in Case 3. Let B be the set of vertices on the blossom, excluding v; let A be the tree path of ancestors of v, including v. (In Edmonds' terminology, A is the *stem* of the blossom.) Let C be an alternating cycle before the shrinking. If $C \cap B = \emptyset$, C exists after the shrinking. Suppose on the other hand that $C \cap B \neq \emptyset$. Let x be the outer vertex in $C \cap B$ that is an ancestor of any other outer vertices in $C \cap B$. There is such a vertex, since if y is an inner vertex in $C \cap B$, then *mate*(x) is an outer vertex in $C \cap B$.) Since x is a proper descendant of v, x is scanned, which implies by Lemma 3.6 that C contains an outer vertex other than x that is an ancestor of x. By the choice of x, any such vertex must be on A. Let y be the vertex in $C \cap A$ that is a descendant of any others. Let D be the part of C starting with the edge $\{y, mate(y)\}$ and continuing from $mate$(y) until reaching a vertex in B. Let E be the part of A connecting y and v. (if $y = v$, E is empty.) The edges corresponding to $D \cup E$ in the graph after shrinking the blossom form an alternating cycle. □

Theorem 3.8 *If there is an alternating cycle in the original graph, then the algorithm halts in Case 2.*

Proof. Suppose to the contrary that the original graph contains an alternating cycle but the algorithm halts after the outer loop. Using induction to apply Lemma 3.7 to each blossom-shrinking, we find that the final graph contains an alternating cycle, say C. In the final graph, all vertices are labeled and all outer vertices are scanned, including all outer vertices on C. Let x be an outer vertex on C such that no outer vertex $y \neq x$ on C is an ancestor of x. This choice of x contradicts Lemma 3.6, which means that the algorithm must halt before finishing the outer loop, namely in Case 2. □

It remains for us to prove Lemma 3.6. To do this, we need two lemmas that express the throughness of the algorithm in eventually applying Cases 2 and 3 to the appropriate arcs.

Lemma 3.9 *Any arc (v, w) with v outer, w inner, and v a descendant of w is never replaced. Furthermore, if such an arc ever exists, the algorithm halts in Case 2 before v is scanned.*

Proof. Let (v, w) be an arc satisfying the hypothesis of the lemma that is not a replacement of an arc satisfying the hypothesis. Then (v, w) is either an original arc or replaces an arc (x, w) with x inner. In either case (v, w) is unmarked when it first exists. Vertex v must be scanned before (v, w) can be replaced, but the scan of v will mark (v, w) and cause the algorithm to halt in Case 2, unless such a halt has previously occurred. □

Lemma 3.10 *Let (v, w) be an arc with both v and w outer and v an ancestor of w. Then (v, w) can be replaced only by an arc of the form (v, x), and (v, w) and all its replacements are deleted before v is scanned.*

Proof. Let (v, w) be an arc satisfying the hypothesis of the lemma that is not a replacement of an arc satisfying the hypothesis. We claim that (v, w) is unmarked when it first exists. For this to be false, (v, w) must replace a marked arc (v, x) with x inner. Since x is a descendant of w, w must be labeled before x, and hence before (v, x) is marked. But if w is labeled before (v, x) is marked, it must also be scanned before (v, x) is marked, which means that (v, w) must replace (v, x) while it is still unmarked, a contradiction.

Since (v, w) is unmarked when it first exists, scanning v will apply Case 3 to (v, w) or some replacement, deleting it, unless w was previously shrunk into v. $\square$

Proof of Lemma 3.6. Let C be an alternating cycle containing a scanned vertex x. We wish to show that C contains an outer vertex $w \neq x$ that is an ancestor of x. Form $\vec{C}$ by directing C in the direction consistent with the arc $(mate(x), x)$. Let (v, w) be the first unmatched arc along $\vec{C}$ starting from x such that either w is outer or $mate(w)$ is not scanned before x. (This includes the case $mate(w) = x$ and the case $mate(w)$ is outer but not scanned.) There is such an arc, because the arc into $mate(x)$ is a candidate. By the choice of (v, w), v is outer, either $v = x$ or v is scanned before x, and $w \neq x$.

We claim that w is an ancestor of both v and x. If w is outer, it cannot be scanned before x by Corollary 3.2 and Lemma 3.10. Let y be the outer vertex in $\{w, mate(w)\}$. Vertex w, and hence y, must be labeled and become active before v is scanned and hence before x is scanned. Since y is not scanned before x, y and hence w must remain active until x is scanned. Since an active vertex is an ancestor of the current vertex being searched, w must be an ancestor of both v and x.

Vertex w must be outer, since by Lemma 3.9 it cannot be inner. Hence it satisfies the lemma. $\square$

3.3 Implementation

Implementation of the algorithm is straightforward except for blossom-shrinking. Consider the replacement history of an arc (r, s) that has a marked replacement. Either r is outer or the first shrinking of r into another vertex x produces a replacement arc (x, y) for (r, s) with x outer. Until marking occurs, subsequent replacement arcs have the same first vertex. This suggests keeping track of replacement arcs semi-implicitly, by making changes to first vertices explicitly and using a disjoint set data structure to track changes in second vertices.

To implement this strategy, we need a data structure to represent disjoint sets subject to two operations: $find(x)$, which returns the name of the set containing x, and $unite(x, y)$, which forms the union of the sets named x and y, naming the new set x and destroying the old sets. (See [12, 28, 29].)

We store the graph vertices in this structure. Each vertex is initially in a self-named singleton set. In Case 3, when shrinking a blossom defined by an arc (v, w), we initialize $x = w$ and repeat the following steps until $x = v$: let $y = p(x)$, perform $unite(y, x)$, and replace y by x.

The algorithm only needs to keep track of unmarked arcs, not marked ones. Initially, we construct a list of arcs incident out from each vertex. To mark an arc out of a vertex v, we get the first arc (v, y) on the arc list of v, delete (v, y) from the list, let $w = find(y)$, and apply the appropriate case to (v, w) if $v \neq w$. (The test $v \neq w$ eliminates arcs deleted by blossom-shrinking.) In Case 3, when shrinking a blossom defined by an arc (v, w), for each inner vertex x on the blossom, we remove each arc (x, y) on the arc list for x and add a corresponding arc (v, y) to the arc list for v. With this implementation, the algorithm runs in $O(m)$ time, not counting set operations. If we use the incremental tree disjoint set union algorithm [12], the set operations take a total of $O(m)$ time as well.

If we want the algorithm to actually produce an alternating cycle as well as just reporting one, we can add a mechanism for expanding blossoms, as described in [28]. Producing such a cycle takes $O(n)$ time once the algorithm halts in Case 2 (see [28]).

3.4 Listing of Matchings

We can apply the alternating-cycle-finding algorithm of this section to the problem of listing all perfect (or maximum cardinality) matchings of a given graph. Itai, Rodeh, and Tanimoto [19] describe how to list all perfect (or maximum-cardinality) matchings in a bipartite graph in $O(m\sqrt{n})$ time to find the first matching plus $O(m)$ time for each subsequent matching. They observe that their algorithm also applies to the nonbipartite case: Once the first matching is found, the time for each subsequent matching is $O(m)$ plus the time to find an alternating cycle. Since by our method the time for the latter is $O(m)$, we obtain from their result an algorithm for listing all perfect (or maximum-cardinality) matchings in $O(m\sqrt{n})$ time for the first matching plus $O(m)$ for each subsequent matching.

4 Unique f-factors

Throughout this section we consider undirected multigraphs. Specifically, we allow parallel edges with arbitrary multiplicities as well as self-loops. A self-loop at a vertex v contributes two to the degree of v. For a multigraph $G = (V, E)$ with a function $f : V \to \mathbf{Z}_+$ (for $\mathbf{Z}_+$ the set of all nonnegative integers) an *f-factor* is a subgraph F whose degree function is given by f, i.e., every vertex v is incident to exactly $f(v)$ edges of F. In this section we extend our results from perfect matchings to f-factors. We also describe how to enumerate f-factors efficiently, further extending the result of Itai, Rodeh, and Tanimoto [19].

We begin with a generalization of Kotzig's Lemma 2.1 proved by Jackson and Whitty [20]. An *alternating cycle* C for an f-factor F is an even-length cycle whose edges are alternately in F and not in F; C is allowed to repeat vertices but not edges. (Note that C can contain two distinct copies of the same edge. Both copies can belong to F, both to $E - F$, or one to each.) Such a cycle gives another f-factor $F \oplus C$ that is distinct from F. Here $\oplus$ denotes symmetric difference.

Let d be the degree function of G.

Theorem 4.1 ([20]) *A multigraph with a unique f-factor and $0 < f(v) < d(v)$ for every vertex v has a bridge.*

We shall give an alternative proof of Theorem 4.1 below. For now we use the theorem to extend the algorithm of Section 2 to decide whether or not a multigraph G with function f has a unique f-factor. If the answer is yes, our algorithm finds the f-factor. The time for the algorithm is $O(m \log^4 n)$. We assume that the multigraph is represented by specifying a numerical multiplicity for each edge. We denote by m the number of distinct edges.

Generalizing Section 2, call a connected component C *even* or *odd* according to the parity of $\sum\{f(v) : v \in C\}$. The definition of *even bridge* and *odd bridge* is unchanged. Note that a bridge must be either even or odd since $\sum\{f(v) : v \in V\}$ is even. A bridge belongs to the desired f-factor if and only if it is odd.

The data structures for our algorithm are as follows. As in Section 2.1, we maintain a set of edges F that belong to the desired f-factor, as well as the current multigraph G and a set of unprocessed bridges R. In addition, for each vertex v we maintain a value $f(v)$ equal to the number of edges in the current multigraph G that must be incident to v in the desired f-factor. (Initially f is the given function.) Also, we maintain $d(v)$ to be the degree of v in the current G. We also maintain the set U of unprocessed vertices v that have $f(v) = d(v)$.

Each iteration of the algorithm processes either a bridge in R or a vertex of U. (The choice of bridge or vertex is arbitrary.) An odd bridge is deleted from G and added to F; the values of d and

f at the ends of the bridge are decreased by 1. In contrast with Section 2, an end v of the bridge and its incident edges are deleted only if $f(v)$ becomes 0. When such an edge (v, w) is deleted, if $w \in U$ we conclude that there is no unique f-factor. Otherwise we decrease $d(w)$ by one and add w to U if this makes $d(w) = f(w)$. We also detect new bridges as in the algorithm of Section 2.1. An even bridge is deleted from G; the value of d at each end v is decreased by 1, and if this makes $f(v) = d(v)$ then v is added to U. A vertex of U is processed by adding each of its incident edges to F, deleting these edges, adjusting d and f as above, and detecting new bridges of G as in the algorithm of Section 2.1. By Theorem 4.1 this algorithm terminates with F being an f-factor and G being a null graph if and only if G has a unique f-factor.

The implementation of Section 2.2 applies to our algorithm with just straightforward modifications. The timing analysis also applies. We conclude that our algorithm tests if a given multigraph with function f has a unique f-factor in time $O(m \log^4 n)$.

Next we extend the algorithm of Section 3 to check if a given f-factor F is unique. Equivalently we find an alternating cycle for F if one exists. We present an algorithm that runs in $O(m)$ time.

The approach is to reduce the problem to the case of matching. That is, we construct a graph G' with a perfect matching M' such that M' is the unique perfect matching of G' if and only if F is the unique f-factor of G. We can decide the former condition using the algorithm of Section 3. Jackson and Whitty use this approach to prove Theorem 4.1 [20]. Even if G has no parallel edges, however, their graph G' can contain $\Theta(nm)$ edges. For efficiency, we require G' to have $O(m)$ edges.

First we deal with a degenerate case. If there are two copies of a multiple edge, with one in F and the other in $E - F$, then F is not unique; the two copies form an alternating cycle. In what follows, we assume that each multiple edge either has all its copies in F or all in $E - F$.

We construct the desired graph G' using the "sparse substitute" technique of [10]. We begin with the key observation of [10]: A shortest alternating cycle in G contains at most four edges incident to any vertex v; two of these edges can be matched and two unmatched. For completeness we prove this. Suppose v is on more than four edges in the alternating cycle C. Traverse the cycle C, starting at v and traversing an edge of F incident to v. The first time we return to v is while traversing an edge of F. (If the edge were in $E - F$ we could shorten the cycle.) Since the cycle alternates we leave v on an edge of $E - F$. The next time we return to v is by traversing an edge of $E - F$ and this is the last edge of C. (Otherwise we could shorten the cycle.)

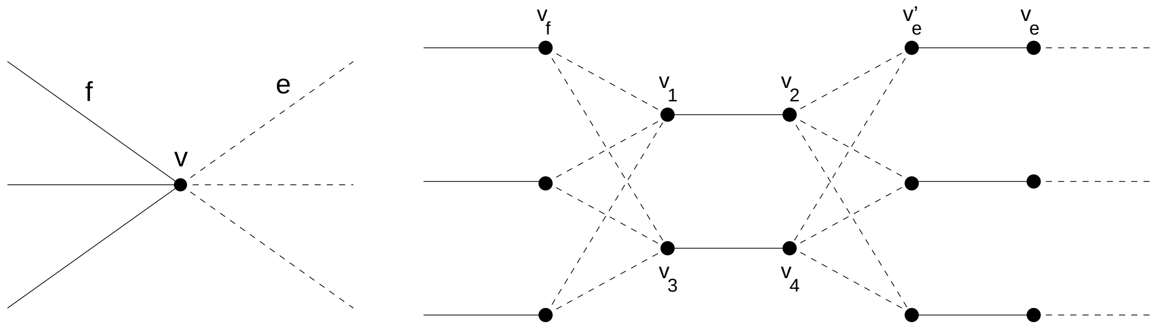

Figure 2: (a) Vertex v in a graph with an f-factor. (b) Sparse substitute for v. Solid lines are in the f-factor (a) and in the matching (b).

Now we describe how to construct G' and corresponding matching M'. The first step is to decrease each edge-multiplicity that is greater than two to two, updating $d(v)$ and $f(v)$ appropriately. By the observation in the previous paragraph, an alternating cycle in the new graph with respect to the new f-factor exists if and only if there was one in the original graph with respect to the original f-factor. (Since copies of a multiple edge are either all in F or all in $E-F$, at most two copies of such an edge can occur on a shortest alternating cycle.) The second step is to replace each vertex v by a sparse substitute. Figure 2 illustrates how to construct a sparse substitute for a vertex v: Create four new vertices v_1, v_2, v_3, v_4 and matched edges $\{v_1, v_2\}$ and $\{v_3, v_4\}$. For each edge $f = \{v, w\} \in F$, create a new vertex v_f and two unmatched edges $\{v_1, v_f\}$ and $\{v_3, v_f\}$. For each edge $e = \{v, w\} \in E - F$, create two new vertices v_e and v'_e, a matched edge $\{v_e, v'_e\}$, and two unmatched edges $\{v_2, v'_e\}$ and $\{v_4, v'_e\}$. Finally, for each edge $e = \{v, w\}$, add an edge $\{v_e, w_e\}$. This edge is matched in G' if and only if $e \in F$.

It is easy to verify that there is a correspondence between alternating cycles in the multigraph G and the graph G': such a cycle in G that contains a vertex v once (twice) corresponds to a cycle in G' containing $\{v_1, v_2\}$ or $\{v_3, v_4\}$ ($\{v_1, v_2\}$ and $\{v_3, v_4\}$) and vice-versa; every alternating cycle in G' can only go through a sparse substitute along one of the two paths of the form v_f, v_1, v_2, v'_e, v_e or v_f, v_3, v_4, v'_e, v_e. (See Figure 2.) It follows that F is the unique f-factor of G if and only if M' is the unique perfect matching of G'.

The sparse substitute for v contains $2 + 2f(v) + 3(d(v) - f(v))$ edges, where $d(v)$ and $f(v)$ are calculated after we reduce the multiplicities to at most two. Hence G' has $O(m)$ edges, as desired. Graph G' can be constructed in $O(m)$ time, and the algorithm of Section 3 runs in $O(m)$ time. In summary, we can find an alternating cycle in G, if one exists, in time $O(m)$.

We can use this algorithm for finding an alternating cycle with respect to a given f-factor to list all f-factors of a given multigraph by observing that the algorithm of Itai, Rodeh, and Tanimoto [19] for listing matchings extends in an entirely straightforward way to the listing of f-factors. The time to find the first f-factor is $O(m\sqrt{F})$ using the algorithm of [10]. The time to find each subsequent f-factor is the time to find an alternating cycle, which is $O(m)$ by our result above.

We conclude this section by sketching how the sparse substitute construction gives a proof of Theorem 4.1. Suppose G has a unique f-factor and every vertex v satisfies $0 < f(v) < d(v)$. Since G' has a unique matching, Kotzig's lemma implies that G' has a matched bridge. The assumption $0 < f(v) < d(v)$ for all vertices implies that the sparse substitute edges $\{v_1, v_2\}$ and $\{v_3, v_4\}$ are on cycles and thus not bridges. If the bridge of G' is a sparse substitute edge $\{v_e, v'_e\}$ then the other edge adjacent to v_e is also a bridge, since v_e has degree two. We conclude that some edge of the form $\{v_e, w_e\}$ is a bridge of G'. It is easy to see that then $\{v, w\}$ must be a bridge of G.

5 Unique Weighted Matchings

Consider an undirected graph $G = (V, E)$ with an edge-weight function $w : E \to \mathbf{R}$. A *maximum-weight perfect matching* is a perfect matching whose edges have the greatest total weight possible. A *maximum-weight matching* is defined similarly but is not required to be perfect. This section gives linear-time algorithms for deciding if a graph has a unique maximum-weight matching or perfect matching, given appropriate information about one such matching. We first discuss the case of perfect matching. We give a precise statement of the problem and then give the algorithm. Then we extend the results to maximum-weight matching.

We use the following additional notation. $\mathbf{R}_+$ denotes the set of all nonnegative real numbers. For a function $f : V \to \mathbf{R}$ and a subset $U \subseteq V$, define $f(U) = \sum\{f(u) : u \in U\}$. A family $\mathcal{F}$ of subsets of V is *laminar* if any two sets of $\mathcal{F}$ are either disjoint or one includes the other.

Consider a graph $G = (V, E)$. For $F \subseteq E$, $V(F)$ denotes the set of all vertices incident to an edge of F. For $U \subseteq V$, $G[U]$ denotes the subgraph of G induced by U.

Suppose $\overline{G}$ is a graph formed by contracting one or more subsets of V. By convention $\overline{G}$ retains parallel edges, i.e., an edge of E is in $\overline{G}$ unless both its ends belong to the same vertex of $\overline{G}$. For an edge $e \in E$, we denote by e also the corresponding edge in $\overline{G}$ (if it exists); similarly for sets of edges.

This notation will cause no confusion.

We now review some basic ideas of Edmonds' algorithm, as applied to the problem of finding a maximum-weight perfect matching [6, 22]. Edmonds showed that a perfect matching M^* has maximum weight if and only if there are "dual functions" $y : V \to \mathbf{R}$ and $z : 2^V \to \mathbf{R}_+$ satisfying the following three conditions: For every edge e,

$$w(e) \leq y(e) + z(\{B : e \subseteq B\}). \tag{1}$$

(Our notational conventions imply that an edge $e = \{v, w\}$ has $y(e) = y(v) + y(w)$.) Let T be the set of all "tight" edges, i.e., edges e satisfying (1) with equality. Then

$$M^* \subseteq T. \tag{2}$$

To state the last condition we first make a definition: A *blossom family* is a laminar family $\mathcal{B}$ of subsets of V where every nonsingleton set $B \in \mathcal{B}$ is called a *blossom* and has the following properties: B is the disjoint union of sets B_i, $i = 1, \ldots, o$, where $o \geq 3$ is odd and the B_i are the maximal proper subsets of B belonging to $\mathcal{B}$. If we contract each B_i there is a set of *odd-cycle edges* $C \subseteq T$ that form a cycle on the contracted vertices B_i. The edges of this cycle C are alternately matched and unmatched, the first and last edges being unmatched.

The third condition for optimality is that there is a blossom family $\mathcal{B}$ such that

$$z(B) > 0 \Longrightarrow B \text{ is a blossom of } \mathcal{B}. \tag{3}$$

A simple argument by induction on $|B|$ shows that a blossom has an odd number of vertices. We will use the following two properties of blossoms, both easily proved by similar inductive arguments:

(i) Any blossom B contains a unique vertex b, called the *base* of B, such that M^* induces a perfect matching on $B - b$.

(ii) For any blossom B and any vertex $v \in B$, T contains a perfect matching on $B - v$. In fact this perfect matching can be chosen to consist of odd-cycle edges of B and all the blossoms contained in B.

We also recall how conditions (1)–(3) imply the optimality of M^*. Let $\mathcal{P}$ (the "positive" blossoms) be the laminar family consisting of all blossoms $B \in \mathcal{B}$ that have $z(B) > 0$. For any perfect matching

M, adding inequality (1) for every edge $e \in M$ gives $w(M^*) \leq y(V) + \sum\{\lfloor |B|/2 \rfloor z(B) : B \in \mathcal{P}\}$. Since every edge of M^* satisfies (1) with equality we have

$$w(M^*) = y(V) + \sum\{\lfloor |B|/2 \rfloor z(B) : B \in \mathcal{P}\}. \tag{4}$$

These relations imply that M^* is a maximum-weight perfect matching.

We shall solve the following problem: Given a graph $G = (V, E)$ with a maximum-weight perfect matching M^*, and given corresponding dual functions y, z and blossom family $\mathcal{B}$ satisfying (1)–(3), we wish to decide if M^* is the unique maximum-weight perfect matching of G. Note that Edmonds' algorithm for maximum-weight perfect matching [6, 22] as well as all known variations (e.g. [13]) produce this information when finding M^*. We give an algorithm to solve this problem in time $O(m)$.

Let $H = (V, T)$ be the graph of tight edges of G. Let $\overline{G}$ be the graph formed from H by contracting every maximal blossom of $\mathcal{P}$. For each blossom $B \in \mathcal{P}$ let $\overline{B}$ be the graph formed from $H[B]$ by contracting every maximal proper subset of B in $\mathcal{P}$. Also let $\widehat{B}$ be the graph $\overline{B}$ with the contracted vertex containing the base of B deleted. Say that a matching is *near-perfect* if exactly one vertex is unmatched.

Lemma 5.1 *Let M^*, y, z and $\mathcal{B}$ satisfy (1)–(3). A perfect matching M on G has maximum weight if and only if it satisfies (2) and for every $B \in \mathcal{P}$, M induces a near-perfect matching on $\overline{B}$.*

Proof. We first show the "if" direction. Let M be a perfect matching satisfying the conditions of the lemma. A simple induction on $|B|$ shows that M contains exactly $\lfloor |B|/2 \rfloor$ edges with both ends in B for every $B \in \mathcal{P}$. Since every edge of M is tight, $w(M)$ is equal to the right-hand side of (4). Hence M has maximum weight.

To show the "only if" direction consider any maximum-weight perfect matching M. (4) and (1) imply that M satisfies (2). (4) also implies that M contains $\lfloor |B|/2 \rfloor$ edges with both ends in B, for every $B \in \mathcal{P}$. Now a simple induction on $|B|$ shows that M induces the near-perfect matchings specified in the lemma. □

Lemma 5.2 *M^* is the unique maximum-weight perfect matching of G if and only if the graphs $\overline{G}$ and $\widehat{B}$ for every blossom $B \in \mathcal{P}$ have a unique perfect matching.*

Proof. The "only if" direction follows using Lemma 5.1 and property (ii) of blossoms.

To prove the "if" direction let M be a maximum-weight perfect matching. M induces a perfect matching on $\overline{G}$ by the second condition of Lemma 5.1 (applied to the maximal blossoms of $\mathcal{P}$). Hence the hypothesis of the lemma shows that M and M^* contain exactly the same edges of $\overline{G}$.

Now we repeat this argument on blossoms $B \in \mathcal{P}$: Suppose we have shown that M and M^* both match the same edge incident to B (and no other edges). M induces a perfect matching on $\widehat{B}$ by the second condition of Lemma 5.1. Hence the hypothesis of the lemma shows that M and M^* agree on the edges of $\widehat{B}$. $\square$

Our algorithm checks if M^* is the unique maximum-weight perfect matching by checking the condition of the lemma, as follows. M^* induces a perfect matching on each of the graphs $\overline{G}$, $\widehat{B}$, $B \in \mathcal{P}$. We check that each of these matchings is unique using the algorithm of Section 3.

To show that this procedure uses total time $O(m)$ note that every invocation uses linear time by Section 3.3. A given edge of E belongs to at most one graph $\overline{G}$, $\widehat{B}$, $B \in \mathcal{P}$. Hence the total time is linear.

In summary, we have shown how to decide if G has a unique maximum-weight perfect matching in time $O(m)$. When the answer is "no", the algorithm can be extended to find another maximum-weight perfect matching in the same time bound. To do this note that the algorithm of Section 3 finds an alternating cycle in some graph $\overline{G}$, $\widehat{B}$. A perfect matching of $\overline{G}$ extends to a maximum-weight perfect matching using property (ii) of blossoms. A perfect matching of $\widehat{B}$ extends similarly. It is a simple matter to construct these extensions in linear time.

We can also use the algorithm to check if a graph has a unique maximum-weight matching. A maximum-weight matching M^* has duals y, z and a blossom family $\mathcal{B}$ satisfying (1)–(3) and the additional condition that for every vertex v,

$$y(v) \geq 0, \text{ and } y(v) > 0 \Longrightarrow v \in V(M^*). \tag{5}$$

Given such a matching M^* on a graph $G = (V, E)$ and given $y, z, \mathcal{B}$, we check if the maximum-weight matching is unique as follows. Construct the graph G' by taking two copies of G and joining the two copies of every vertex $v \in V$ by an edge of weight zero. Construct a perfect matching M' on G' by matching the two copies of each edge of M^*, as well as the weight zero edge corresponding to every unmatched vertex of M^*. Use the given y, z and $\mathcal{B}$ on both copies of G in G'. It is easy to see that

(1)–(3) hold for G' (in particular if $v \in V$ is unmatched, the corresponding weight-zero edge of M' is tight because of (5)). Furthermore M^* is the unique maximum-weight matching of G if and only if M' is the unique maximum-weight perfect matching of G'. Hence our original algorithm can check uniqueness.

The algorithms of this section are useful for sensitivity analysis in settings where a unique maximum-weight (perfect) matching is desired. (See Section 1.) A maximum-weight (perfect) matching can be found from scratch in time $O(n(m+n\log n))$ [15], or $O(\sqrt{n\alpha(m,n)\log n}\, m\log(nN))$ when edge weights are integers between 0 and N [13]. The matching (and its corresponding duals and blossom family) can be updated faster, however. For instance, if arbitrary changes are made to the edges incident to one vertex, a new maximum-weight (perfect) matching (with duals and blossom family) can be constructed by finding one weighted augmenting path [3, 14]. This can be done in time $O(m + n\log n)$ [15]. After such an update, our algorithm can be used to test uniqueness of the new maximum-weight (perfect) matching, in additional time $O(m)$.

Acknowledgments

We thank Tibor Jordán for pointing out reference [20]. The second author thanks Martin Golumbic for motivating discussions. We also thank the anonymous referees for their perceptive suggestions.

References

[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 270–280, 1997.

[2] S. Alstrup, J. Holm, and M. Thorup. Maintaining center and median in dynamic trees. In *Proc. 7th Scandinavian Workshop on Algorithm Theory (SWAT)*. Springer-Verlag, 2000.

[3] M.O. Ball and U. Derigs. An analysis of alternative strategies for implementing matching algorithms. *Networks*, 13:517–549, 1983.

[4] T. C. Biedl, P. Bose, E. D. Demaine, and A. Lubiw. Efficient algorithms for Petersen's matching theorem. In *Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 130–139, 1999.

[5] R.B. Carey and G.D. Stormo. Graph-theoretic approach to RNA modeling using comparative data. In *Proceedings 3rd International Conf. on Intelligent Systems for Molecular Biology*, pages 75–80, 1995.

[6] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *J. Res. Nat. Bur. Standards*, 69B:125–130, 1965.

[7] J. Edmonds. Paths, trees, and flowers. *Canadian J. Math*, pages 233–240, 1965.

[8] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing*, 14(4):781–798, 1985.

[9] H. N. Gabow. Algorithmic proofs of two relations between connectivity and the 1-factors of a graph. *Discrete Math*, 26:33–40, 1979.

[10] H. N. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *Proc. 15th Annual ACM Symp. on Theory of Comp.*, pages 448–456, 1983.

[11] H. N. Gabow, H. Kaplan, and R. E. Tarjan. Unique maximum matching algorithms. In *Proc. 31st Annual ACM Symp. on Theory of Comp.*, pages 70–78. ACM, 1999.

[12] H. N. Gabow and R. E. Tarjan. A linear time algorithm for a special case of disjoint set union. *J. Computer and System Sciences*, 30:209–221, 1985.

[13] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph matching problems. *J. Assoc. Comput. Mach.*, 38:815–853, 1991.

[14] H.N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proc. 26th Annual Symp. on Found. of Comp. Sci.*, pages 90–100, 1985.

[15] H.N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. 1st Annual ACM-SIAM Symp. on Disc. Algorithms*, pages 434–443, 1990.

[16] D. Giammarresi and G. F. Italiano. Decremental 2- and 3- connectivity of planar graphs. *Algorithmica*, 16:263–287, 1996.

[17] M. C. Golumbic, T. Hirst, and M. Lewenstein. Uniquely restricted matchings. manuscript, 1998.

[18] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. 30th Annual ACM Symp. on Theory of Comp.*, pages 79–89, 1998.

[19] A. Itai, M. Rodeh, and S.L. Tanimoto. Some matching problems for bipartite graphs. *J. ACM*, 25(4):517–525, 1978.

[20] B. Jackson and R. W. Whitty. A note concerning graphs with unique f-factors. *J. Graph Theory*, 13:577–580, 1989.

[21] A. Kotzig. On the theory of finite graphs with a linear factor I. *Mat.-Fyz. Časopis Slovensk. Akad. Vied*, 9:73–91, 1959.

[22] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Reinhart and Winston, New York, 1976.

[23] L. Lovász and M.D. Plummer. *Matching Theory.* North-Holland, Amsterdam, 1986.

[24] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

[25] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Computer and System Sciences*, 26(3):362–391, 1983.

[26] J.E. Tabaska, R.B. Cary, H.N. Gabow, and G.D. Stormo. An RNA folding method capable of identifying pseudoknots and base triples. *Bioinformatics*, 14(8):691–699, 1998.

[27] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.

[28] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, 1982.

[29] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. Assoc. Comput. Mach.*, 31:245–281, 1984.

[30] Mikkel Thorup. Decremental dynamic connectivity. *J. Algorithms*, 33(2):229–243, Nov 1999.