Unique Maximum Matching Algorithms

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

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. 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. Also, 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. We prove a generalization of Kotzig's theorem characterizing unique f-factors in terms of bridges. This theorem 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.

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. [12]. 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 each vertex exactly once.)

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 [14] 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 [13]. An improvement in dynamic connectivity testing, specifically for 2-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 [2]. This gives an efficient constructive proof of Petersen's theorem, which states that any such graph has a perfect matching.

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. [12].) (An *alternating cycle* is a simple cycle alternating edges in M, called *matched edges*, with edges not in M, called *unmatched edges*.) This algorithm, described in Section 3, combines Edmonds' classical blossom-shrinking algorithm for finding a maximum-cardinality matching [5] with depth-first search.

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. In Section 4 we prove a generalization of Kotzig's theorem characterizing unique f-factors in terms of bridges. Specifically, we 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 that tests whether a given graph has a unique f-factor. The running time

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

†AT&T Labs Research, Florham Park, NJ. e-mail: hkl@research.att.com.

‡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.

remains $O(m \log^4 n)$.

We 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 [7] for sparsifing the resulting graph, allows us to adapt our second algorithm to test whether a given f-factor is unique in linear time.

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 [4] 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 [3] and [17] 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 matches 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. Note that in either case the component originally containing the bridge must be even.

Lemma 2.1 *Suppose G has a unique perfect matching M. Then every component of G is even, every odd bridge of G is in M, and every even bridge of G is not in M.*

Proof. By parity. □

The usefulness to us of this observation comes from the following converse:

Lemma 2.2 (Kotzig [14]; see also [6, 16]) *A unique perfect matching contains a bridge.*

Applied recursively, Lemmas 2.1 and 2.2 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.

Lemma 2.3 *Each edge added to M is an odd bridge in the current graph. The final graph contains no odd bridges.*

Proof. Obvious. □

Theorem 2.4 *G has a unique perfect matching, namely M, if and only if the final graph is empty.*

Proof. Immediate from lemmas 2.1, 2.2, and 2.3 by induction. □

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-component*.

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-components,
        find a path $P(v, w)$ connecting v and w
         and add every bridge on $P(v, w)$ to R.

Lemma 2.5 *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-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. □

Corollary 2.6 *Theorem 2.4 holds for the refined algorithm.*

Proof. Immediate. □

2.2 Implementation

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 [13]. The dynamic

2-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-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 [13], although they are also easy to implement in the 2-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-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 [13].

The data structure of [13] represents a tree T in a spanning forest of G by a *top-tree* $\mathcal{T}$. Each node of a top-tree represents a *cluster*. A cluster has one or two *boundary nodes*. A cluster with boundary nodes a, b represents a subtree of T that contains both a and b. If vertices x and y are in T, let $P(x, y)$ denote the path in T from x to y. A cluster with boundary nodes a and b has *cluster path* $P(a, b)$.

We begin by performing the operation $expose(v, w)$. This operation modifies the top-tree $\mathcal{T}$ so that the root cluster has boundary nodes v and w. The cluster path $P(v, w)$ 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.

After performing $expose(v, w)$, we traverse one or more paths in $\mathcal{T}$, each going from the root of $\mathcal{T}$ to a leaf. These leaves will be precisely the bridges on $P(v, w)$. (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. (A *path-cluster* is a cluster with two distinct boundry nodes, i.e., $a \neq c$.) 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-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-components.

It remains to indicate how we check that a and b are in different 2-components. This is easy to do because 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-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.

2.3 Efficiency

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 $\mathcal{T}$. By definition 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 its ends deleted, or is a bridge of the final graph. Furthermore, as noted in the proof of Lemma 2.5, 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 $O(m \log^4 n)$ [13]. This gives the desired time bound.

For the case of planar graphs, we can improve the running time to $O(m \log n)$. We omit the description of this result here. For the case of dense graphs, we can improve the running time by up to a polylogarithmic factor. We note first that Thorup [22] has proposed a Las Vegas randomized algorithm for deletion-only connectivity that has a running time of $O(m \log(n^2/m) + (nm)^{1/2} \log^{2.5} n) = O(n^2)$ [21]. 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 begining 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 bridge components 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 [18], 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 $G' - F_1$. We omit here the straightforward proof that the graph induced by S has the same bridges and 2-edge components as G'.

We initialize F_1 to be any spanning forest of G, and we use the dynamic connectivity algorithm of [13], 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 bridges 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.

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 [18] 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 [5], it occasionally shrinks a blossom (an odd-length simple cycle alternating matched and unmatched edges, with one extra unmatched edge). 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 [9] 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 in Section 3.3. Section 3.4 provides details of the implementation of blossom-shrinking. Throughout, we assume familiarity with depth-first search, as described e.g. in [18], and with Edmonds' algorithm [5], as described e.g. in [19].

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. Shrinking does not change the inner/outer status of any vertex. The vertices being combined form a path in the spanning forest from a vertex v to 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 replacement can create multiple arcs, which we allow. An arc (x, y) with both x and y on the path is simply 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.

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.) 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 from w to v plus the arc (v, w) defines a blossom. Shrink this blossom to the single vertex v.

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

We must eventually 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.

3.2 Properties of the Algorithm

First, we make some easy observations concerning blossom-shrinking. Vertices are only shrunk into outer vertices that are their ancestors. Blossom-shrinking preserves the ancestor-descendant relationship. Vertices can be shrunken into a given vertex v only while v is the vertex currently being scanned.

Second, we explicate the meaning of "active".

Lemma 3.1 *A vertex is active if and only if it is an ancestor of the vertex v currently being scanned.*

Proof. By induction on the number of steps taken by the algorithm. The only interesting case is Case 3, in which a path of descendants of the current vertex v is shrunk into it. All vertices on this path except v are inactive when the shrinking occurs (their scans are completed). Such a shrinking maintains the invariant. □

Third, we prove a structural lemma about arcs, which among other things implies that Case 3 never halts with an error.

Lemma 3.2 *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 is labeled.*

Proof. Let (v, w) be an arc such that v is outer, w is labeled, and v was labeled before w. We must show that w is a descendant of v, which is true if w is labeled before v is scanned.

Assume w is labeled after v is scanned. When v is scanned there must exist an arc (v, z) with either $z = w$ or z a descendant of w. Furthermore since v is scanned arc (v, z) must be marked and z must be labeled. But since w is an ancestor of z, w must be labeled by this time also, a contradiction. □

Corollary 3.3 *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 such that v and w are outer. Then (w, v) is such an arc as well. If v was labeled before w, Lemma 3.2 applied to (v, w) implies that w is a descendant of v. If on the other hand w was labeled before v, Lemma 3.2 applied to (w, v) implies that v is a descendant of w. □

Corollary 3.4 *Case 3 of the algorithm can never halt with an error.*

Proof. Let (v, w) be an arc that triggers Case 3. Both v and w are outer, and since w is inactive, w cannot be an ancestor of v. By Corollary 3.3 w must be a descendant of v. □

Fourth, we prove two lemmas expressing the thoroughness of the algorithm in eventually applying Cases 2 and 3 when there exist appropriate arcs.

Lemma 3.5 *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.*

Proof. Let (v, w) be an arc satisfying the hypothesis of the lemma that is not a replacement of an arc satisfying the lemma. 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 3.6 *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 any replacement) is 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 an 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 shrunken into v. □

Fifth and finally, we prove that blossom-shrinking and blossom-expansion preserve the existence of alternating cycles. To verify the former, we need a lemma about the structure of alternating cycles.

Lemma 3.7 *Let C be an alternating cycle containing a scanned vertex x. Then C contains a vertex w that is an outer proper ancestor of x.*

Proof. 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 and either $v = x$ or v is scanned before x.

We claim that w is an ancestor of both v and x. If w is outer, it is scanned after x by Lemma 3.6. 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. By Lemma 3.1, w is an ancestor of both v and x.

By Lemma 3.5, w cannot be inner. Thus it must be outer, and it cannot equal x. □

Corollary 3.8 *If there is an alternating cycle before a blossom-shrinking, then there is one after.*

Proof. Consider shrinking a blossom formed by an arc (v, w) in Case 3. Let B be the set of vertices on the blossom, and let P be the tree path of ancestors of v (including v). Let C be an alternating cycle. If C misses $B - \{v\}$, C exists after the blossom-shrinking. Suppose, on the other hand, that C hits $B - \{v\}$. Let x be the outer vertex in $C \cap (B - \{v\})$ that is an ancestor of any others. Since x is a proper descendant of v, x is scanned, which by Lemma 3.7 implies that C contains an outer proper ancestor of x. By the choice of x, any such vertex must be on P. Let y be the vertex in $C \cap P$ 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 on B. Let E be the part of P connecting y and v (may be empty). The edges corresponding to $D \cup E$ in the shrunken graph form an alternating cycle. □

Lemma 3.9 *If there is an alternating cycle after blossom-shrinking, then there is one before.*

Proof. As in Corollary 3.8, consider shrinking a blossom when marking an arc (v, w) in Case 3. Let B be the set of vertices on the blossom. 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 $x \neq v$ in B; D contains the edge $\{mate(v), v\}$ and no vertices in B except x and v. 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. □

3.3 Correctness

We now have enough machinery that the proof of correctness is easy.

Theorem 3.10 *The algorithm halts in Case 2 if and only if there is an alternating cycle in the original graph.*

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. By Lemma 3.9 and induction on the number of blossom-shrinkings, there is an alternating cycle in the original graph.

Conversely, suppose there is an alternating cycle in the original graph. By Corollary 3.8 and induction on the number of blossom-shrinkings, there is always an alternating cycle. Suppose the algorithm halts in the outer loop. All vertices are labeled and all outer vertices are scanned. Let C be an alternating cycle in the final graph. Let x be an outer vertex on C that has no proper ancestor on C. By Lemma 3.7, x has a proper ancestor on C. This contradiction implies that the algorithm cannot halt in the outer loop and therefore halts in Case 2. □

3.4 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 [8, 19, 20].) 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 [8], 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 [19]. Producing such a cycle takes $O(n)$ time once the algorithm halts in Case 2 (see [19]).

4 Unique f-factors

Throughout this section we consider undirected multigraphs, i.e., parallel edges with arbitrary multiplicities are allowed, as well as self-loops. A self-loop (v, v) contributes two to the degree of v. For a multigraph $G = (V, E)$ with a nonnegative-valued function $f : V \to \mathbf{Z}$, 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. This section extends our results from perfect matchings to f-factors.

We begin by generalizing Kotzig's Lemma 2.2. Recall that 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. 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 *A multigraph with a unique f-factor and $0 < f(v) < d(v)$ for every vertex v has a bridge.*

Proof. The argument is by reduction to the case of perfect matching. Let $G = (V, E)$ be a multigraph satisfying the hypothesis of the theorem, with a unique f-factor F. Construct a graph G' with a perfect matching M' as follows. Graph G' depends on f-factor F. For each vertex $v \in V$, G' has $d(v)$ *external vertices* and $d(v) - f(v)$ *internal vertices*. Collectively these $2d(v) - f(v)$ vertices are the *substitute vertices* of v. For each edge $(v, w) \in E$, G' has an edge (v', w'), where v' (w') is an external vertex of v (w). These $d(v)$ edges are the *external edges* of v. Each external vertex of v is on exactly one external edge. For each edge $(v, w) \in F$, we call both v' and w' *externally matched*, where (v', w') is the edge of G' corresponding to (v, w). Each of the $d(v) - f(v)$ external vertices corresponding to v that is not externally matched has an edge in G' connecting it to an internal vertex corresponding to v; each internal vertex is used exactly once. Finally

each internal vertex corresponding to v is connected to each external vertex corresponding to v that is externally matched. Each edge incident on an internal vertex of G' is called an internal edge.

The perfect matching M' of G' consists of the external edges of G' corresponding to the edges of F plus the internal edges joining internal vertices with external vertices that are not externally matched.

The construction makes M' a perfect matching on G'. Now we show that M' is the unique perfect matching. Suppose the contrary. This implies there is an alternating cycle C' (for matching M'). It is easy to see that the edges of C' form the following pattern that repeats every six edges: an unmatched internal edge and a matched internal edge, both joining substitute vertices of v, followed by an unmatched external edge (v', w'), followed by a matched internal edge and an unmatched internal edge, both joining substitute vertices of w, followed by a matched external edge (w', z'). Construct an alternating cycle for F by letting the above six edges correspond to edges $(v, w), (w, z)$, two edges not in F and in F respectively. The existence of this alternating cycle contradicts the hypothesis that F is the unique f-factor. We conclude that M' is the unique perfect matching of G'.

Lemma 2.2 implies G' has a bridge (a, b), separating G' into connected components A and B. Consider any vertex $v \in V$. The hypothesis $0 < f(v) < d(v)$ implies that the substitute vertices of v induce a connected subgraph. Hence all substitute vertices of v are in the same component A or B, unless a substitute vertex of v is on the bridge.

This fact implies that if an external edge (v', w') is a bridge of G' then (v, w) is a bridge of G. To complete the proof we show that G' always has an external edge that is a bridge. Assume that the given bridge (a, b) is an internal edge, say (v', v'') with v' external and v'' internal. Let the component containing v' (v'') when the bridge is removed be A' (A'').

First suppose (v', v'') is matched. Every substitute vertex of v except v' belongs to A''. This follows since $f(v) > 0$. Now it is easy to see that the (unmatched) external edge incident to v' is a bridge.

Next suppose (v', v'') is unmatched. (Hence v' is externally matched.) A' contains every internal vertex of v, and its mate, other than v'. A'' contains every remaining external vertex of v, i.e., every externally matched vertex other than v'. Since (v', v'') is a bridge there cannot be both an internal vertex other than v'' and an externally matched vertex other than v'. If v'' is the only internal vertex then the (matched) external edge incident to v' is the desired bridge. If v' is the only externally matched vertex then v'' and its mate, say ν', form a bridge. Furthermore the (unmatched) external edge incident to ν' is the desired bridge. □

Now we sketch how 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.

The algorithm extends the algorithm of 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. A bridge belongs to the desired f-factor if and only if it is odd.

The data structures for our algorithm are as follows. We maintain for each vertex v the value $f(v)$ to be the number of edges in the current multigraph G that must be incident to v in the desired f-factor. We also maintain $d(v)$ to be the degree of v in the current multigraph. (Both of these quantities may be decreased by the algorithm when edges are deleted.) There is a queue Q of vertices v that have $f(v) = 0$ or $f(v) = d(v)$.

The algorithm is similar to the one in Section 2. Each iteration deletes either a bridge or a vertex from G. (The vertex is one removed from Q.) In contrast with Section 2 the ends of an odd bridge are not immediately deleted. (They eventually get deleted when they are removed from Q.) A vertex is deleted in a manner similar to the original algorithm.

We conclude this section by sketching how to decide whether or not a multigraph G with f-factor F has F as its unique f-factor. The time for this algorithm is $O(m)$.

The proof of Theorem 4.1 constructs a graph G' with perfect matching M' such that F is the unique f-factor of G if and only if M' is the unique perfect matching of G'. We can decide the latter condition using the algorithm of Section 3. However even if G has no parallel edges G' can contain $\Theta(nm)$ edges. It suffices to show how to construct a graph H with perfect matching M, such that H has an alternating cycle if and only if G' does, and H has $O(m)$ edges.

We follow the approach of [7]. There it is shown that a shortest alternating cycle in G' contains at most four external edges for any vertex v (two of these edges can be matched and two unmatched). Thus we can assume each multiplicity in G is at most two. Furthermore [7] shows that in G' the internal edges of v can be replaced by $3d(v) - f(v) + 2$ edges without changing the set of alternating cycles. To do this we use a "sparse substitute." Using a sparse substitute for each vertex of G gives the desired graph H with $O(m)$ edges. Note also that H can be constructed in $O(m)$ time.

5 Unique Maximum-Weight Matchings

Consider an undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbf{R}$. A *maximum-weight matching* is a matching whose edges have the greatest total weight possible. This section discusses the problem of deciding if a graph has a unique maximum-weight matching, given appropriate information about one such matching. Our first goal is to state this problem precisely.

We use the following additional notation. $\mathbf{R}_+$ denotes the set of nonnegative real numbers. For a function $f : V \to \mathbf{R}$ and a subset $U \subseteq V$, $f(U) = \sum\{f(u) : u \in U\}$. 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. Let $\overline{G}$ be 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 the corresponding edge in $\overline{G}$ (if it exists) also by e; similarly for sets of edges. This will cause no confusion.

We now review some basic ideas of Edmonds' algorithm [4] (see also [15]) for finding a maximum-weight matching. Edmonds showed that a 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 four conditions: For every edge e,

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

Let T be the set of all "tight" edges, i.e., edges e satisfying (1) with equality.

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

$$y(v) > 0 \Longrightarrow v \in V(M^*) \tag{3}$$

To state the fourth condition we need 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 each B_i is a maximal subset of B in $\mathcal{B}$. If we contract each B_i there is a set of edges $C \subseteq T$ that forms a cycle on the vertices B_i. The edges of cycle C are alternately matched and unmatched, with the first and last edges unmatched.

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

$$z(B) > 0 \Longrightarrow B \text{ is a blossom of } \mathcal{B}. \tag{4}$$

A simple induction shows that a blossom has an odd number of vertices. We will use the following two properties of blossoms, both easily proved by similar inductions:

(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 edges in the above-mentioned cycles that define B and the blossoms contained in B.

This section solves the following problem: Given a graph $G = (V, E)$ with a maximum-weight matching M^*, along with corresponding dual functions y, z and blossom family $\mathcal{B}$, we wish to decide if M^* is the unique maximum-weight matching of G. Note that Edmonds' algorithm for maximum-weight matching [4, 15] as well as all known variations (e.g. [9]) produce this information when finding M^*. We give an algorithm to solve this problem in time $O(m)$.

Let $\mathcal{P}$ (the "positive" blossoms) be the laminar family consisting of all blossoms $B \in \mathcal{B}$ with $z(B) > 0$. Define the graph of tight edges $H = (V, T)$. Let $\overline{G}$ be H with every blossom of $\mathcal{P}$ contracted. For any blossom $B \in \mathcal{P}$ let $\overline{B}$ be the graph formed from $H[B]$ by contracting every maximal subset of B in $\mathcal{P}$. Also let $\overline{B^-}$ be the graph $\overline{B}$ with the vertex containing the base of B deleted. Finally say that a matching is *near-perfect* if exactly one vertex is unmatched.

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

Proof. It is well-known and easy to see that

$$w(M^*) = y(V) + \sum\{\lfloor |B|/2 \rfloor z(B) : B \in \mathcal{P}\}. \tag{5}$$

Any matching satisfying the conditions of the lemma has maximum weight, by (5).

To show the converse consider any maximum-weight matching M. Conditions (5) and (1) imply that M satisfies (2) and (3). Condition (5) also implies that M contains $\lfloor |B|/2 \rfloor$ edges with both ends in B, for every $B \in \mathcal{P}$. A simple induction shows that M induces the near-perfect matchings described in the lemma. □

Let U be the set $V - V(M^*)$ of unmatched vertices of M^*. Let $\overline{U}$ denote the images of these vertices in $\overline{G}$ (i.e., if $u \in U$ is in a maximal blossom B of $\mathcal{P}$, replace u by B.) Let Z be the set of all vertices $v \in V$ with $y(v) = 0$. Let $\overline{Z}$ denote the images of these vertices in $\overline{G}$. Finally, define a *proper alternating path* to be an alternating path that starts (ends) with an unmatched edge if and only if the first (last) vertex is unmatched. (The path may start and/or end with a matched edge.)

Lemma 5.2 *M^* is the unique maximum-weight matching of G if and only if the following four conditions hold:*

(a) $\overline{G}$ has no proper alternating path joining two distinct vertices of $\overline{Z}$;
(b) $\overline{G}$ does not contain an alternating cycle for M^*;
(c) every blossom in $\overline{U}$ contains a unique vertex of Z;
(d) for each blossom $B \in \mathcal{P}$, $\overline{B^-}$ has a unique perfect matching.

Proof. The necessity of (a) – (d) follows using Lemma 5.1 and property (ii) of blossoms.

To prove sufficiency, consider any maximum-weight matching M. M induces a matching on $\overline{G}$ by Lemma 5.1. Conditions (a) and (b) along with (3) imply that M and M^* agree on edges of $\overline{G}$. Conditions (c) and (d), and a simple induction, imply that M and M^* agree inside any blossom of $\mathcal{P}$. □

Condition (d) can be checked using the algorithm of Section 3: for each blossom B, one perfect matching of $\overline{B^-}$ is given by the corresponding edges of M^*. The total time to check every blossom $B \in \mathcal{P}$ is $O(m)$, since an edge of E is in at most one graph $\overline{B^-}$. Condition (b) is checked similarly. Condition (c) is trivial to check. To check condition (a), for each matched vertex $v \in \overline{Z}$ add a new vertex v' and an unmatched edge $\{v, v'\}$ to $\overline{G}$. Now check if the new graph has an augmenting path. (No edge weights are involved here.) This takes linear time, as described in [19].

In summary, we have shown how to decide if G has a unique maximum-weight matching in time $O(m)$. When the answer is "no", a simple extension of the algorithm can find another maximum-weight matching, in the same time bound.

Our algorithm is useful for sensitivity analysis in settings where a unique maximum-weight matching is desired. (See Section 1.) A maximum-weight matching can be found from scratch in time $O(n(m+n\log n))$ [11], or $O(\sqrt{n\alpha(m,n)\log n}\ m\log(nN))$ when edge weights are integers between 0 and N [9]. However, a maximum-weight matching (with duals and blossom family) can be updated faster, e.g., if arbitrary changes are made to the edges incident to one vertex, a new maximum-weight matching (with duals and blossom family) can be constructed by finding one weighted augmenting path, in time $O(m+n\log n)$ ([1, 10]). After such an update, our algorithm can be used to test uniqueness of the new maximum-weight matching, in additional time $O(m)$.

Acknowledgments

The second author thanks Martin Golumbic for motivating discussions.

References

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

[2] 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.

[3] 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.

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

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

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

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

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

[17] 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.

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

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

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

[21] M. Thorup. Private communication, 1998.

[22] M. Thorup. Decremental dynamic connectivity. In *Proc. 8th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 305–313, 1997.