Data Structures for Weighted Matching and Extensions to *b*-matching and *f*-factors

HAROLD N. GABOW, University of Colorado at Boulder

This article shows the weighted matching problem on general graphs can be solved in time $O(n(m + n \log n))$ for n and m the number of vertices and edges, respectively. This was previously known only for bipartite graphs. The crux is a data structure for blossom creation. It uses a dynamic nearest-common-ancestor algorithm to simplify blossom steps, so they involve only back edges rather than arbitrary nontree edges.

The rest of the article presents direct extensions of Edmonds' blossom algorithm to weighted b-matching and f-factors. Again, the time bound is the one previously known for bipartite graphs: for b-matching the time is $O(\min\{b(V), n \log n\}(m + n \log n))$ and for f-factors the time is $O(\min\{f(V), m \log n\}(m + n \log n))$, where $b(V)$ and $f(V)$ both denote the sum of all degree constraints. Several immediate applications of the f-factor algorithm are given: The generalized shortest path structure of Reference [19], i.e., the analog of the shortest-paths tree for conservative undirected graphs, is shown to be a version of the blossom structure for f-factors. This structure is found in time $O(|N|(m + n \log n))$ for N, the set of negative edges ($0 < |N| < n$). A shortest T-join is found in time $O(n(m + n \log n))$ or $O(|T|(m + n \log n))$ when all costs are nonnegative. These bounds are all slight improvements of previously known ones, and are simply achieved by proper initialization of the f-factor algorithm.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; *Data structures design and analysis*;

Additional Key Words and Phrases: Matching, *b*-matching, *f*-factor, blossom, degree-constrained subgraph, shortest-paths tree, conservative graph, T-join

ACM Reference format:
Harold N. Gabow. 2018. Data Structures for Weighted Matching and Extensions to *b*-matching and *f*-factors. *ACM Trans. Algorithms* 14, 3, Article 39 (June 2018), 80 pages.
https://doi.org/10.1145/3183369

1 INTRODUCTION

This article solves a well-known problem in data structures to achieve an efficient algorithm for weighted matching. It also extends the results to the most general weighted matching problems. This section defines the problems and states the results.

A *matching* on a graph is a set of vertex-disjoint edges. A matching is *perfect* if it covers every vertex. More generally, it is *maximum cardinality* if it has the greatest possible number of edges,

This work was supported in part by the National Science Foundation under Grant No. CCR-8815636.
This article is a combination of two conference papers: A preliminary version of the data structures part appeared in *Proceedings of the 1st Annual ACM-SIAM Symp. on Disc. Algorithms*, 1990 [15]. A preliminary version of the extensions part, based on reduction to matching, appeared in *Proceedings of the 15th Annual ACM Symp. on Theory of Comp.*, 1983 [13].
Author's address: H. N. Gabow, University of Colorado at Boulder, Department of Computer Science, Boulder, CO, 80309-0430; email: hal@cs.colorado.edu.

https://doi.org/10.1145/3183369

and *cardinality* k if it has exactly k edges. Let each edge e have a real-valued *weight* $w(e)$. The weight $w(S)$ of a set of edges S is the sum of the individual edge weights. Each of the preceding matching variants has a maximum weight version, e.g., a *maximum weight matching* has the greatest possible weight, a *maximum weight perfect matching* has maximum weight subject to the constraint that it is *perfect*, and so on. Alternatively, edges may have real-valued costs, and we define *minimum cost matching* and the like. The *weighted matching problem* is to find a matching of one of these types, e.g., find a maximum weight perfect matching on a given graph and the like. All these variants are essentially equivalent from an algorithmic viewpoint. For definiteness this article concentrates on maximum weight perfect matching.

In stating resource bounds for graph algorithms we assume throughout this article that the given graph has n vertices and m edges. For notational simplicity we assume $m \geq n/2$. In the weighted matching problem, this can always be achieved by discarding isolated vertices.

Weighted matching is a classic problem in network optimization; detailed discussions are in References [6, 27, 28, 30, 34]. Edmonds gave the first polynomial-time algorithm for weighted matching [9]. Several implementations of Edmonds' algorithm have been given with increasingly fast running times: $O(n^3)$ [11, 27], $O(mn \log n)$ [4, 22], $O(n(m \log \log \log_{2+m/n} n + n \log n))$ [17]. Edmonds' algorithm is a generalization of the Hungarian algorithm, due to Kuhn, for weighted matching on bipartite graphs [25, 26]. Fredman and Tarjan implement the Hungarian algorithm in $O(n(m + n \log n))$ time using Fibonacci heaps [10]. They ask if general matching can be done in this time. We answer affirmatively: We show that a search in Edmonds' algorithm can be implemented in time $O(m + n \log n)$. This implies that the weighted matching problem can be solved in time $O(n(m + n \log n))$. The space is $O(m)$. Our implementation of a search is in some sense optimal: As shown by Fredman and Tarjan [10] for Dijkstra's algorithm, one search of Edmonds' algorithm can be used to sort n numbers. Thus a search requires time $\Omega(m + n \log n)$ in an appropriate model of computation.

Weighted matching algorithms based on cost-scaling have a better asymptotic time bound when costs are small integers [21]. However, our result remains of interest for at least two reasons: First, Edmonds' algorithm is theoretically attractive because its time bound is strongly polynomial. Second, for a number of matching and related problems, the best known solution amounts to performing one search of Edmonds' algorithm, e.g., most forms of sensitivity analysis for weighted matching [4, 8, 14, 38]. Thus our implementation of a search in time $O(m + n \log n)$ gives the best-known algorithm for these problems.

The article continues by presenting versions of Edmonds' blossom algorithm for weighted b-matching and weighted f-factors. These problems generalize ordinary matching to larger degree-constrained subgraphs and are defined as follows. For an undirected multigraph $G = (V, E)$ with function $f : V \to \mathbb{Z}_+$, an *f-factor* is a subgraph where each vertex $v \in V$ has degree exactly $f(v)$. For an undirected graph $G = (V, E)$, where E may contain loops, with function $b : V \to \mathbb{Z}_+$, a (perfect) *b-matching* is a function $x : E \to \mathbb{Z}_+$, where each vertex $v \in V$ has $\sum_{w:vw \in E} x(vw) + x(vv) = b(v)$. Given, in addition, a weight function $w : E \to \mathbb{R}$, a *maximum b-matching* is a (perfect) b-matching with the greatest weight possible; similarly for *maximum f-factor*. We find maximum b-matchings and f-factors in the same time bound as was known for bipartite graphs: for b-matching the time is $O(\min\{b(V), n \log n\}(m + n \log n))$, where $b(V)$ is the sum of all degree constraints; for f-factors the time is $O(\min\{f(V), m \log n\}(m + n \log n))$, where $f(V)$ is the sum of all degree constraints. A blossom algorithm for b-matching is given in Pulleyblank's thesis [32] (Reference [33] gives a very high-level description, different from our algorithm). The pseudo-polynomial parts of the above bounds (i.e., the bounds using $b(V)$ and $f(V)$) can also be achieved using the current article's algorithm for ordinary matching plus the reduction to matching presented in the original version of the current article [13].

Here we prefer direct implementations of the general matching algorithms to avoid practical inefficiencies and to illuminate the properties of blossoms. As an example of the latter, the algorithm's blossom structure is shown to be exactly the generalized shortest path structure of Gabow and Sankowski [19], i.e., the analog of the shortest-paths tree for conservative undirected graphs. (The paths in blossoms that are used to augment the matching give the shortest paths to the fixed source in a conservative undirected graph.) Our discussion of blossoms also leads to (and requires) simple proofs of (previously known) min-max formulas for the maximum size of a b-matching, Equation (4.3), or partial f-factor, Equation (5.6). Lastly. our algorithm shows that b-matching blossoms have the same structure as ordinary matching blossoms (unlike f-factors there are no "$I(B)$-sets," i.e., pendant edges, and no "heavy blossoms," only "light" ones, at least in the final output).

We find the generalized shortest path structure in time $O(|N|(m + n \log n))$ for N the set of negative edges ($0 < |N| < n$ for conservative costs) and a shortest T-join in time $O(n(m + n \log n))$, or $O(|T|(m + n \log n))$ for nonnegative costs. These bounds are slight improvements of previously known ones and are achieved simply by proper initialization of the f-factor algorithm. (The strong polynomial bound of Gabow and Sankowski [19, Section 10] can be modified to achieve the same time as ours for the matching part, plus additional time $O(m \log n)$ for post-processing.) Good implementations of the T-join algorithm of Edmonds use time $O(n^3)$ for general costs and the same time as ours plus $O(|T|^3)$ for nonnegative costs, both cubic terms coming from finding a minimum cost matching on a complete graph [34, p. 486 and p. 488].

The article is organized as follows. This section concludes with some terminology and assumptions. Section 2 reviews Edmonds' algorithm and defines the "blossom-merging problem"—the last ingredient needed to obtain the time bound we seek. Section 3 specializes this problem to "tree-blossom-merging" and solves it. Section 4 gives our b-matching algorithm and Section 5 gives the f-factor algorithm. Appendix A gives further details of Edmonds' matching algorithm. Appendix B gives some further details for b-matching and f-factors. Appendix C gives a data structure to compute alternating paths in matching blossoms, and alternating trails for b-matching and f-factors. Appendix D gives an efficient implementation of the grow and expand steps of Edmonds' algorithm. Gabow [14] gives a faster algorithm, but our approach is simpler and suffices to achieve the overall time bound. The appendix also gives extensions to b-matching and f-factors. Appendix E shows the generalized shortest path structure of Reference [19] exists for real-valued edge costs.

Our algorithm for tree-blossom-merging requires an algorithm that computes nearest common ancestors in trees that grow by addition of new leaves. The conference version of this article [15] presented the required algorithm, as well as extensions. For reasons of succinctness, the data structure for nearest common ancestors is now given separately in Reference [16].

Portions of this article may be read independently. Our implementation of Edmonds' matching algorithm is in Section 3; readers familiar with Edmonds' algorithm can skip the review and go directly to Section 2.3. Those interested in generalized versions of matching should concentrate on f-factors, our most general algorithm (Section 5), although some basic lemmas are proved in Section 4.

History of This Article. The conference paper [15] presented a preliminary version of the tree-blossom-merging algorithm. The current article simplifies that algorithm, e.g., there is no need to refine the strategy for sparse graphs ($m = o(n \log^2 n)$). The tree-blossom-merging algorithm uses an algorithm that computes nearest common ancestors in trees that grow by addition of new leaves. Reference [15] presented the required algorithm as well as extensions. As mentioned, this is now given in Reference [16]. Subsequent to Reference [15], Cole and Hariharan [5] used a similar approach to allow other operations; they also achieve time bounds that are worst-case rather than amortized.

The results on b-matching and f-factors evolve from the conference paper [13]. That paper achieved similar time bounds to those presented here by reducing the problems to matching. The current article gives direct approaches to the problems, thus illuminating the structure of blossoms (see Sections 4.1 and 5.1) and avoiding the blow-up in problem size incurred by reduction.

Terminology. We often omit set braces from singleton sets, denoting $\{v\}$ as v. We use interval notation for sets of integers: for $i, j \in \mathbb{Z}$, $[i..j] = \{k^{\in\mathbb{Z}} : i \le k \le j\}$. We use a common summing notation: If x is a function on elements and S a set of elements, then $x(S)$ denotes $\sum_{s\in S} x(s)$. $\log n$ denotes logarithm to the base two. Assume that for a given integer $s \in [1..n]$, the value $\lfloor \log s \rfloor$ can be computed in $O(1)$ time. This can be done if we precompute these n values and store them in a table. The precomputation time is $O(n)$.

For a graph G, $V(G)$ denotes its vertices and $E(G)$ its edges. For vertices x, y an *xy-path* has ends x and y. For a set of vertices $S \subseteq V$ and a subgraph H of G, $\delta(S, H)$ ($\gamma(S, H)$) denotes the set of edges with exactly one (respectively, two) endpoint in S. (Loops are in γ but not δ.) $d(v, H)$ denotes the degree of vertex v in H. When referring to the given graph G, we often omit the last argument and write, e.g., $\delta(S)$. (For example, a vertex v has $d(v) = |\delta(v)| + 2|\gamma(v)|$.)

Fix a matching M on the given graph. A vertex is *free* if it is not on any matched edge. An *alternating path* is a vertex-simple path whose edges are alternately matched and unmatched. (Paths of 0 or 1 edge are considered alternating.) An *augmenting path* P is an alternating path joining two distinct free vertices. To *augment the matching along* P means to enlarge the matching M to $M \oplus P$ (the symmetric difference of M and P). This gives a matching with one more edge.

2 EDMONDS' ALGORITHM AND ITS IMPLEMENTATION

This section summarizes Edmonds' algorithm and known results on its implementation. Sections 2.1–2.2 sketch the high level algorithm. They include all the details needed for our implementation but do not give a complete development. For the latter, see, e.g., References [9, 11, 27, 30]. Section 2.3 reviews the portions of the algorithm for which efficient implementations were previously known, and the outstanding problem, efficient "blossom-merging."

2.1 Blossoms

Throughout this section, the notation $P(x, y)$ denotes an xy-path. This includes the possibility that $x = y$, i.e., $P(x, x) = (x)$.

Edmonds' algorithm is based on the notion of blossom. We start with a data-structure-oriented definition, illustrated in Figure 1. Begin by considering two even-length alternating paths $P(x_i, y)$, $i = 0, 1$, with $x_0 \ne x_1$ and y the only common vertex. Each path begins with the matched edge at x_i, unless $x_i = y$. These paths plus an edge x_0x_1 form a simple example of a blossom. Edmonds' algorithm contracts blossoms. This leads to the general, recursive definition:

Definition 2.1. Let G be a graph with a distinguished matching. A *blossom* is a subgraph defined by rules (a) and (b):

(a) Any single vertex b is a blossom.

(b) Let $\overline{G}$ be a graph formed from G by contracting zero or more vertex-disjoint blossoms. Let X_0, X_1, Y be $\overline{G}$-vertices, $X_0 \ne X_1$. For $i = 0, 1$ let $P(X_i, Y)$ be an even-length alternating path that starts with a matched edge or has $X_i = Y$, with Y the only common $\overline{G}$-vertex. These paths plus an edge X_0X_1 form a blossom.

In Figure 1, blossom B is formed from paths $P(B_4, B_0)$, $P(B_6, B_0)$, and edge γ. We use the term "vertex" to refer to a vertex of the given graph G. For a blossom B, $V(B)$ is the set of vertices of G

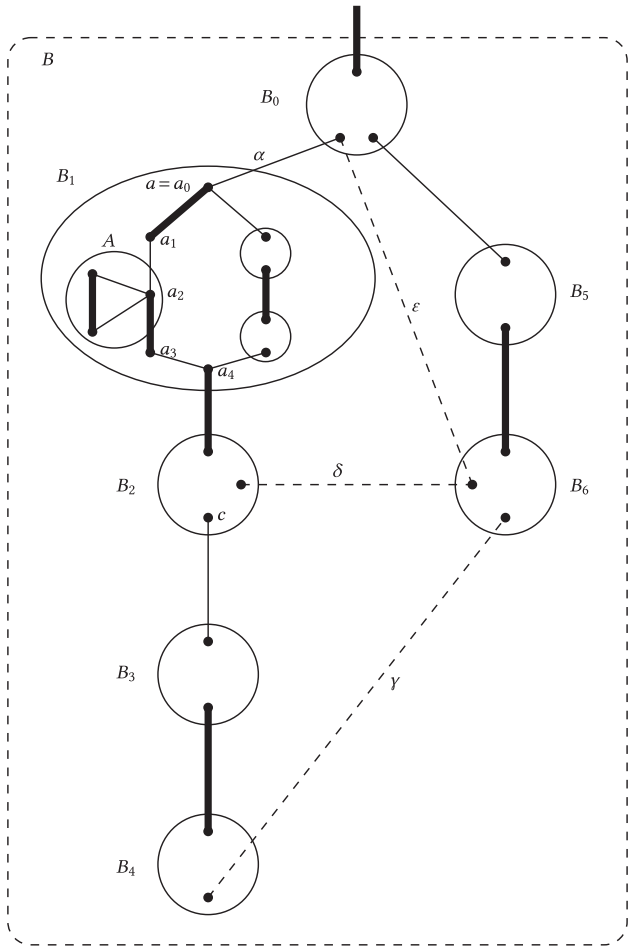

Fig. 1. Blossoms in a search of Edmonds' algorithm. Blossom B is formed from subblossoms $B_0, \ldots, B_6$. Heavy edges are matched.

contained in any blossom in either path $P(X_i, Y)$; we sometimes refer to them as *the vertices of B*. The maximal blossoms in the paths $P(X_i, Y)$ are *the subblossoms of B*.

We use some properties of blossoms that are easily established by induction. Any blossom has a *base vertex*: In Definition 2.1, a blossom of type (a) has base vertex b. A blossom of type (b) has the same base vertex as Y. We usually denote the base of B as $\beta(B)$ (or β if the blossom is clear). $\beta(B)$ is the unique vertex of B that is not matched to another vertex of B. $\beta(B)$ is either free or matched to a vertex not in B. We call B a *free blossom* or *matched blossom* accordingly.

Consider a blossom B with base vertex β. Any vertex $x \in V(B)$ has an even-length alternating path $P(x, \beta)$ that starts with the matched edge at x. ($P(\beta, \beta)$ has no edges.) For example, in Figure 1 in blossom B, $P(a, \beta(B))$ starts with $P(a, \beta(B_1)) = (a_0, a_1, a_2, a_3, a_4)$ followed by edge $(\beta(B_1), \beta(B_2))$ and the reverse of $P(c, \beta(B_2))$, and continuing along paths in B_3, B_4, B_6, B_5, B_0.

To define $P(x, \beta)$ in general, consider the two paths for B in Definition 2.1. Among the blossoms in these paths let x belong to a blossom designated as B_0. The edges of B (i.e., the edges of $P(X_i, Y)$ plus X_0X_1) contain a unique even-length alternating path A from B_0 to Y. Here A is a path in $\overline{G}$.

First, suppose $B_0 \neq Y$. A starts with the matched edge at B_0. $P(x, \beta)$ passes through the same blossoms as A. To be precise, let the $\overline{G}$-vertices of A be $B_i, i = 0, \ldots, k$, $B_k = Y$, with $k > 0$ even. Let β_i be the base vertex of B_i. So there are vertices $x_i \in V(B_i)$ such that the edges of A are

$$\beta_0\beta_1, x_1x_2, \beta_2\beta_3, x_3x_4, \ldots, x_{k-1}x_k.$$

Here the $\beta_i\beta_{i+1}$ edges are matched and the x_ix_{i+1} edges are unmatched. Recursively define $P(x, \beta)$ as the concatenation of $k + 1$ subpaths

$$P(x, \beta) = P(x, \beta_0), P(\beta_1, x_1), P(x_2, \beta_2), P(\beta_3, x_3), \ldots, P(x_k, \beta_k). \tag{2.1}$$

For odd i, $P(\beta_i, x_i)$ is the reverse of path $P(x_i, \beta_i)$.

Now consider the base case $B_0 = Y$. If Y is a vertex, then $Y = x = \beta$ and $P(x, \beta) = (x)$. Otherwise, $P(x, \beta)$ in blossom B is identical to $P(x, \beta)$ in B_0.[1]

Edmonds' algorithm finds an augmenting path $\overline{P}$ in the graph $\overline{G}$ that has every blossom contracted. $\overline{P}$ corresponds to an augmenting path P in the given graph G: For any contracted blossom B on an unmatched edge xy ($x \in V(B)$) of $\overline{P}$, P traverses the path $P(x, \beta(B))$. If we augment the matching of G along P, every blossom becomes a blossom in the new matched graph: For instance, the above vertex x becomes the new base of B. In Figure 1, if $\overline{P}$ contains an unmatched edge α' that enters B at vertex a, P contains the subpath $P(a, \beta(B))$. The augment makes a the base of B as well as B_1; in the contracted graph α' is the matched edge incident to B.

2.2 Edmonds' Weighted Matching Algorithm

For definiteness, consider the problem of finding a maximum weight perfect matching. The algorithm is easily modified for all the other variants of weighted matching. Without loss of generality assume a perfect matching exists.

The algorithm is a primal-dual strategy based on Edmonds' formulation of weighted matching as a linear program (see the review in Appendix A). It repeatedly finds a maximum weight-augmenting path and augments the matching. The procedure to find one augmenting path is a *search*. If the search is successful, i.e., it finds an augmenting path P, then an *augment step* is done. It augments the matching along P. The entire algorithm consists of $n/2$ searches and augment steps. At any point in the algorithm, $V(G)$ is partitioned into blossoms. Initially, every vertex is a singleton blossom.

Figure 2 gives pseudocode for the search for an augmenting path. A more detailed discussion with examples follows. Assume the graph has a perfect matching so the augmenting path exists. For any vertex v, B_v denotes the maximal blossom containing v.

A search constructs a subgraph $\mathcal{S}$. $\mathcal{S}$ is initialized to contain every free blossom. It is enlarged by executing three types of steps, called *grow*, *blossom*, and *expand steps* in Reference [14]. In addition, the search changes the linear programming dual variables in *dual adjustment steps*. After a dual adjustment step, one or more of the other steps can be performed. Steps are repeated until $\mathcal{S}$ contains the desired augmenting path.

$\mathcal{S}$ consists of blossoms forming a forest. More precisely, if each blossom is contracted to a vertex, $\mathcal{S}$ becomes a forest $\overline{\mathcal{S}}$ whose roots are the free blossoms. A blossom of $\mathcal{S}$ that is an even (odd)

[1] The $P(x, \beta)$ paths may intersect in nontrivial ways. For instance, in Figure 1, $P(\beta(B_3), \beta(B))$ and $P(\beta(B_5), \beta(B))$ traverse edge γ in opposite directions. So, the paths have common subpaths, e.g., the subpath of $P(\beta(B_5), \beta(B))$ joining γ and edge $(\beta(B_3), \beta(B_4))$, and disjoint subpaths, e.g., the subpath of $P(\beta(B_5), \beta(B))$ joining $\beta(B_3)$ and edge α. This intersection pattern can continue inside blossom B_0. So, in general, for two vertices x_0, x_1 in a blossom B with base β, the paths $P(x_i, \beta)$ can have arbitrarily many subpaths that are alternately common and disjoint.

make every free vertex or free blossom the (outer) root of an $\overline{\mathcal{S}}$-tree
loop
  if $\exists$ *tight edge* $e = xy$, x *outer and* $y \notin \mathcal{S}$ **then**
    /* grow step */
    let β be the base of B_y, with $\beta\beta' \in M$
    add $xy, B_y, \beta\beta', B_{\beta'}$ to $\mathcal{S}$
  else if $\exists$ *tight edge* $e = xy$, x, y *outer in the same search tree,* $B_x \neq B_y$ **then**
    /* blossom step */
    merge all blossoms in the fundamental cycle of e in $\overline{\mathcal{S}}$
  else if $\exists$ *tight edge* $e = xy$, x, y *outer in different search trees* **then**
    /* augment step */
    /* xy plus the $\overline{\mathcal{S}}$-paths to x and y form an augmenting path P */
    augment the matching along P and end the search
  else if $\exists$ *a nonsingleton inner blossom* B *with* $z(B) = 0$ **then**
    /* expand step */
    let $\overline{\mathcal{S}}$ contain edges xy and $\beta\beta'$ incident to B where β is the base of B, $x \in V(B)$
    let B have subblossoms B_i
    in $\overline{\mathcal{S}}$ replace B by the even length alternating path of subblossoms $B_0, \ldots, B_k$
      that has $x \in B_0$, $\beta \in B_k$
    /* the remaining subblossoms of B are no longer in $\mathcal{S}$ */
  else adjust duals

Fig. 2. Pseudocode for a search in Edmonds' algorithm.

distance from a root of $\overline{\mathcal{S}}$ is *outer* (*inner*). (The third possibility is a blossom not in $\mathcal{S}$.) A vertex of $\mathcal{S}$ is outer or inner depending on the blossom that contains it. Any path from a root to a node in its tree of $\overline{\mathcal{S}}$ is alternating. So the matched edge incident to a nonroot outer blossom goes to its parent; the matched edge incident to an inner blossom goes to its unique child. In Figure 1, if $B_0, \ldots, B_6$ are maximal blossoms and blossom B has not yet formed, $\overline{\mathcal{S}}$ may contain the solid edges, with all B_i descending from B_0 and B_i outer for i even.

Now we discuss the three steps that build up $\mathcal{S}$. (After that we discuss the role of dual variables and the dual adjustment step.) We illustrate the discussion with Figure 1, again supposing $B_0, \ldots, B_6$ are maximal blossoms.

Grow Step: A grow step enlarges $\overline{\mathcal{S}}$ by adding a new inner blossom B_y and a new outer blossom $B_{\beta'}$. (Each of these blossoms may simply be a vertex of $V(G)$.) Note that when xy is scanned, B_y is guaranteed to be a matched blossom – it is not free by the initialization of $\mathcal{S}$.

In Figure 1, if $\mathcal{S}$ contains outer blossom B_0 but no other B_i, a grow step for edge α adds α, B_1, $(\beta(B_1), \beta(B_2))$ and B_2 to $\mathcal{S}$. Two more grow steps add the other B_i, $i = 3, \ldots, 6$.

Blossom Step: A blossom step contracts the fundamental cycle of e into a new blossom B of $\overline{\mathcal{S}}$. Observe that B is outer: In proof, let A be the blossom closest to the root in e's fundamental cycle. B will be outer if A is outer. If e is incident to A, then $A = B_x$, so A is outer. If e is not incident to A, then the ends of e descend from 2 distinct children of A. A must be outer, since, as previously noted, an inner blossom has only one child.

The blossom step changes some inner vertices to outer. The previously outer vertices remain outer.

In Figure 1, suppose $\mathcal{S}$ contains blossoms B_i, $i = 0, \ldots, 6$, with B_i outer for i even. A blossom step for γ would form the outer blossom B. The B_i with i odd change from inner to outer. Alternatively, the search might do a blossom step for ε, then later one for δ, and still later one for γ. Other sequences are possible, and the one executed depends on the weights of these edges.

Expand Step: An expand step replaces the inner blossom B by an alternating path of one or more blossoms B_i. The B_i are the maximal blossoms along the path $P(x, \beta)$ of B. The remaining subblossoms of B (possibly none) are no longer in $\overline{\mathcal{S}}$ – they are now eligible to enter $\overline{\mathcal{S}}$ in grow steps.

$\overline{\mathcal{S}}$ remains an alternating forest: Some vertices of B change from inner to outer, the others remain inner. Blossoms previously descending from B do not change inner/outer status.

In Figure 1, when $\mathcal{S}$ contains all blossoms B_i, an expand step for B_1 replaces it by vertices a_0, a_1, blossom A, a_3, and a_4. The other two subblossoms of B_1 leave $\overline{\mathcal{S}}$. A subsequent expand step for A does not add any outer vertices.

This completes the description of the three steps that construct $\mathcal{S}$. Note that once a vertex becomes outer it remains in $\mathcal{S}$ and outer for the rest of the search. In contrast vertices can alternate between being inner and not being in $\mathcal{S}$ (perhaps ultimately becoming outer in a grow or blossom step). This alternation can occur $\Theta(n)$ times for a given vertex v in one search. (The upper bound n holds since each time v is involved in an expand step, the size of B_v, the maximal blossom containing v, decreases.)

The search terminates in an augment step. The augmenting path P for $e = xy$ is found in $\overline{\mathcal{S}}$ by tracing the paths from B_x and B_y to their respective tree roots. As mentioned previously, P corresponds to an augmenting path P_G in G, that traverses the blossoms of P along $P(x, \beta)$ paths. The augment rematches the edges of P_G, and the blossoms of P remain blossoms in the rematched graph.

After an augment, the algorithm halts if every vertex is matched. Otherwise, the next search is initialized by making every free blossom a tree root of $\mathcal{S}$. The initialization retains all the blossoms—a maximal blossom that is not free becomes a blossom not in $\mathcal{S}$.[2] (Because of this an implementation of Edmonds' algorithm has a choice for augment steps: It can rematch the complete augmenting path P_G or just the path P in the contracted graph. If the latter, the complete matching on G is computed after the last search, again by explicitly rematching the $P(x, \beta)$ paths.)

We turn to the role of edge weights. The algorithm maintains linear program dual variables (see Appendix A). An edge is *tight* if its dual variables satisfy the linear program's complementary slackness conditions. As such it may be included in a maximum weight matching. Observe that $\mathcal{S}$ always consists entirely of tight edges. So each augment preserves complementary slackness, and the final matching is guaranteed to have maximum weight.

The dual adjustment step modifies dual variables so that additional edges become tight, and corresponding grow, blossom or expand steps can be done. The dual adjustment step involves finding a candidate edge that is closest to being tight, and changing the duals to make it tight. The details are in Appendix A.

2.3 The Blossom-Merging Problem

This completes the sketch of Edmonds' algorithm. Our task is to implement a search in time $O(m + n \log n)$. It is known how to implement most parts of the algorithm within this time bound.

[2]This contrasts with maximum cardinality matching, where initialization discards the blossoms. As a result, there are no nonsingleton inner blossoms.

We review these known parts and then define the remaining implementation problem, blossom merging.

The outer blossoms of $\overline{\mathcal{S}}$ are tracked using an algorithm for set-merging. For instance, a blossom step does *finds* to identify the outer vertices in the fundamental cycle in $\overline{G}$ and *unions* to contract the cycle. It suffices to use an algorithm that executes $O(m)$ *finds* and $O(n)$ *unions* in a universe of n elements in $O(m + n \log n)$ total time and $O(n)$ space (e.g., the simple relabel-the-smaller-set algorithm [1, 7]). The incremental-tree set-merging algorithm of Gabow and Tarjan [20] improves this bound to linear time.

The processing associated with grow and expand steps can be done in time $O(m\alpha(m, n))$ using a data structure for list splitting given in Gabow [14].[3] A simpler algorithm that suffices for our time bound is presented in Appendix D. The current blossom containing a given vertex is found using the above set-merging data structure for outer vertices and the grow/expand algorithm for nonouter vertices.

Blossoms are also represented in a data structure for computing $P(x, \beta)$ paths. These paths are used to find augmenting paths in G and to compute the supporting forest for tree-blossom merging (Section 3). The data structure is also used in expand steps to update $\overline{\mathcal{S}}$. Appendix C gives details for such a data structure. A similar data structure given by Gabow [12] allows all processing of $P(x, \beta)$ paths in a search to be done in time $O(n)$.

For dual adjustment steps, a Fibonacci heap $\mathcal{F}$ is used. It contains the candidate edges for tightness mentioned previously. The heap minimum gives the next edge to be made tight. This is analogous to Fredman and Tarjan's implementation of Dijkstra's algorithm by Fibonacci heaps and uses the same time per search, $O(m + n \log n)$.

Dual variables are maintained in time $O(n)$ per search using offset quantities (e.g., Reference [22]). This leaves only the blossom steps: implementing the blossom steps of a search in time $O(m + n \log n)$ gives the desired result. (This observation is also made by Gabow et al. [17].)

The problem of implementing the blossom steps of a search can be stated precisely as the *blossom-merging problem*, which we now define. (Reference [17] defines a similar problem called on-line restricted component merging, solving it in time $O(m \log \log \log_{2+m/n} n + n \log n)$.) The universe consists of a graph with vertex set $\mathcal{O}$ and edge set $\mathcal{E}$, both initially empty. ($\mathcal{O}$ will model the set of outer vertices, $\mathcal{E}$ the set of edges joining two $\mathcal{O}$-vertices and thus candidates for blossoms.) At any point in time, $\mathcal{O}$ is partitioned into subsets called blossoms. The problem is to process (online) a sequence of the following types of operations:

make_blossom(A) – add the set of vertices A to $\mathcal{O}$ and make A a blossom
(this assumes $A \cap \mathcal{O} = \emptyset$ before the operation);
merge(A, B) – combine blossoms A and B into a new blossom
(this destroys the old blossoms A and B);
make_edge(vw, t) – add edge vw, with cost t, to $\mathcal{E}$
(this assumes $v, w \in O$);
min_edge – return an edge vw of $\mathcal{E}$ that has minimum cost subject to
the constraint that v and w are (currently) in distinct blossoms.

Let us sketch how these four operations are used to implement a search. Grow, expand, and blossom steps each create new outer blossoms. They perform *make_blossom* operations to add the new outer vertices to $\mathcal{O}$. They also perform *make_edge* operations for the new edges that join two outer vertices. For example, in Figure 1, if B_0, B_5 and B_6 are in $\mathcal{S}$ and a grow step is done for edge α then *make_blossom*(B_2) is done; also *make_edge* is done for edge δ.

[3] After the conference version of the paper, the time for list splitting was improved by Pettie [31] and Thorup [37].

Note that in the operation *make_edge*(vw, t), t is the given weight of vw modified by dual values in a way that *min_edge* will give the first blossom step edge to become tight. This modification allows the algorithm to make dual adjustments efficiently (see e.g., Gabow et al. [22]). The value of t is unknown until the time of the *make_edge* operation. (The value is derived from the current slack in the complementary slackness condition for vw; details on this slack are in Appendix D.) From now on, since we are only concerned with the blossom-merging problem, the "cost" of an edge of $\mathcal{E}$ refers to this value t, not the edge weight in the given graph.

A blossom step performs *merges* to construct the new blossom. In Figure 1, the operations *merge*(B_i, B_0), $i = 1, \ldots, 6$ construct B. Note that information giving the edge structure of blossoms (the blossom data structure, the $\overline{\mathcal{S}}$ forest) is maintained and used in the outer part of the algorithm—it is not relevant to the blossom-merging problem. For this problem, a blossom B is identical to its vertex set $V(B)$; the *merge* operation need only update the information about the partition of $\mathcal{O}$ induced by blossoms. Also, in the blossom-merging problem, "blossom" refers to a set of the vertex partition, i.e., the result of a *make_blossom* or *merge* operation. The latter may be only a piece of a blossom in Edmonds' algorithm (as in the six merges above) but this is not relevant.

A *min_edge* operation can be done at the end of each of the three search steps. The returned edge, say e, is used in the above-mentioned Fibonacci heap $\mathcal{F}$ that selects the next step of the search. Specifically, $\mathcal{F}$ has one entry that maintains the smallest cost edge of $\mathcal{E}$. If that entry already contains e, nothing is done. If the entry contains an edge of larger cost than e, the entry is updated to e and a corresponding *decrease_key* is done. The smallest key in $\mathcal{F}$ (which may or may not be the key for e) is used for the next dual adjustment and the next step of the search.

To illustrate this process in Figure 1, suppose $\mathcal{O}$ consists of B_i for $i = 0, 1, 2, 5, 6$ and $\mathcal{E} = \{\delta, \varepsilon\}$. Furthermore, the entry in $\mathcal{F}$ contains edge δ. A grow step for B_3 and B_4 adds $V(B_4)$ to $\mathcal{O}$ and *make_edge* is done for γ. If γ is now the smallest edge in $\mathcal{E}$, the entry in $\mathcal{F}$ for the next blossom step changes from δ to γ and a corresponding *decrease_key* is performed. The next step of the algorithm will be this blossom step, or a grow step for some other edge in $\mathcal{F}$ of smaller cost, or an expand.

Our task is to implement a sequence of these operations: *make_blossoms* adding a total of $\le n$ vertices, $\le m$ *make_edges*, $\le n$ *merges* and $\le n$ *min_edges*, in time $O(m + n \log n)$. The bound on *min_edges* follows since *min_edge* need be done only after a step that makes a vertex of $V(G)$ outer, i.e., after a grow or blossom step, or an expand that creates a new outer vertex. (Recall the expansion of blossom A in Figure 1—no *min_edge* need be done.)

The difficulty in solving the blossom-merging problem is illustrated by Figure 1. When each B_i, $i = 0, \ldots, 6$ is a blossom, edges γ, δ, ϵ are candidates for *min_edge*. If a blossom step for γ is done, δ and ϵ become irrelevant—they no longer join distinct blossoms. If we store the edges of $\mathcal{E}$ in a priority queue useless edges like δ, ϵ can end up in the queue. These edges eventually get deleted from the queue but the deletions accomplish no useful work. The time bound becomes $\Omega(m \log n)$. (This also indicates why there is no need for a *delete_min* operation in the blossom-merging problem: If edge γ gets returned by *min_edge* and as above a blossom step forms B, edge γ becomes irrelevant.)

3 TREE-BLOSSOM-MERGING

Tree-blossom-merging incorporates the topology of the search graph into general blossom-merging, in two ways. This section starts by defining the tree-blossom-merging problem and showing how it can be used to implement the blossom steps of Edmonds' algorithm. Then it presents our tree-blossom-merging algorithm.

The first goal is to maintain a representation of the search graph $\mathcal{S}$ by a forest that changes as little as possible. We cannot avoid adding nodes, e.g., in grow steps. But we can define a forest that does not change in expand steps. Consider a search tree, i.e., a tree $\overline{\mathcal{T}}$ in the forest $\overline{\mathcal{S}}$.

Definition 3.1. A tree T *supports the search tree* $\overline{\mathcal{T}}$ if each blossom B of $\overline{\mathcal{T}}$ has a corresponding subtree T_B in T, these subtrees partition the vertices of T and are joined by edges of $\overline{\mathcal{T}}$, and for each blossom B:

Case B *is outer*: Let B have base vertex β. $V(B) = V(T_B)$. If B is incident to the matched edge $\beta\beta'$ in $\overline{\mathcal{T}}$ then β' is the parent of β in T. If B is a free vertex then β is the root of T.

Case B *is inner*: Let B be incident to edges $vx, \beta\beta'$ in $\overline{\mathcal{T}}$, where $x, \beta \in V(B)$ and $\beta\beta'$ is matched. Then T_B is the path $P(x, \beta)$ and v is the parent of x in T.

The *supporting forest* consists of a supporting tree for each tree of $\overline{\mathcal{S}}$.

Take any vertex v in an outer blossom B of $\overline{\mathcal{T}}$. v has a path to the root in both $\overline{\mathcal{T}}$ and T, say $p_{\overline{\mathcal{T}}}(v)$ and $p_T(v)$, respectively. Let $p_{\overline{\mathcal{T}}}(v) = (B_0, B_1, \ldots, B_k)$ for $B_0 = B$. $p_T(v)$ consists of subpaths through each subtree T_{B_i}. For even i, the subpath contains the base vertex $\beta(B_i)$ and perhaps other B_i-vertices. For odd i, the subpath is the entire path T_{B_i}. This correspondence will allow us to track potential blossom steps, as well as execute them, in the supporting forest.

We will maintain the supporting tree T using this operation:

add_leaf(x, y) – add a new leaf y, with parent x, to T
(x is a node of T, y is a new node not in T).

We assume the data structure for T records parent pointers created by *add_leaf*.

We now show how T is maintained as the search algorithm executes grow, blossom, and expand steps. We use a set-merging algorithm to maintain the partition of $V(T)$ into outer blossoms and individual vertices in inner blossoms.

Suppose a grow step enlarges $\mathcal{S}$ by adding unmatched edge vx, inner blossom B, matched edge $\beta\beta'$ and outer blossom B', where vertices $v \in V(\overline{\mathcal{T}})$, $x, \beta \in B$ and $\beta' \in B'$. First compute $P(x, \beta)$ and write it as x_i, $i = 0, \ldots, k$ where $x_0 = x$, $x_k = \beta$. Enlarge T by performing *add_leaf*(v, x_0), *add_leaf*(x_i, x_{i+1}) for $i = 0, \ldots, k-1$, *add_leaf*(x_k, β') and finally *add_leaf*(β', w) for every $w \in B' - \beta'$. Merge the vertices of B' into one set.

Consider a blossom step for edge vw. It combines the blossoms on the fundamental cycle C of vw in $\overline{\mathcal{T}}$. Let blossom A be the nearest common ancestor of the blossoms containing v and w in $\overline{\mathcal{T}}$. A is an outer blossom. C consists of the subpaths of $p_{\overline{\mathcal{T}}}(v)$ and $p_{\overline{\mathcal{T}}}(w)$ ending at A. In T, merge every outer blossom in C into the blossom A. For each inner blossom B in C, do *add_leaf*(v, u) for every vertex $u \in B - V(T)$. Then merge every vertex of B into A. The new blossom A has the correct subgraph T_A so the updated T supports $\overline{\mathcal{T}}$.

Lastly, consider an expand step. The expand step in the search algorithm replaces an inner blossom B by the subblossoms along the path $P(x, \beta)$, say subblossoms B_i, $i = 0, \ldots, k$, where $x \in B_0$ and $\beta \in B_k$. By definition T_B is the path $P(x, \beta)$. For odd i, B_i is a new outer blossom of $\overline{\mathcal{T}}$. Perform *add_leaf*$(\beta(B_i), v)$ for every vertex $v \in B_i - V(T)$. Merge the vertices of B_i.

For correctness, note that for even i, B_i is a new inner blossom of $\overline{\mathcal{T}}$. Equation (2.1) gives the subpaths of $P(x, \beta)$ through the B_i. T contains the path $P(x_i, \beta(B_i))$, and x_{i-1} is the parent of x_i. So T_{B_i} satisfies Definition 3.1 as required.

This completes the algorithm to maintain T. The total time for maintaining T in a search is $O(n)$. This holds because $P(x, \beta)$ paths are computed in grow steps in time linear in their size, and this allows the outer vertices in expand steps to be identified. Details are in Appendix C.

As mentioned, the correspondence between paths p_T and $p_{\overline{\mathcal{T}}}$ allows us to track potential blossom steps in the supporting forest. This still appears to be challenging. Our second simplification of the blossom-merging problem is to assume *make_edge* adds only back edges, i.e., edges joining a vertex to an ancestor in T. Clearly, a blossom step for an edge vw is equivalent to blossom steps for the two edges va and wa, for a the nearest common ancestor of v and w in T. So, we can replace vw by these two back edges.

To accomplish this reduction, we use a data structure for dynamic nearest common ancestors. Specifically, the algorithm maintains a tree subject to two operations. The tree initially consists of a dummy root r, and it grows using *add_leaf* operations. The second operation is

$nca(x, y)$ – return the nearest common ancestor of vertices x and y.

In summary, we implement Edmonds' search algorithm as follows. The search constructs the supporting forest of $\overline{\mathcal{S}}$ using *add_leaf* operations. (Each supporting tree is rooted at a child of r.) When the search discovers an edge $vw \in E$ joining two outer vertices, it performs $nca(v, w)$ to find the nearest common ancestor a. It executes two blossom-merging operations, $make\_edge(va, t)$ and $make\_edge(wa, t)$ for appropriate t. (Edges with $a = r$ correspond to a future augmenting path.)

Each grow, blossom, and expand step performs all the appropriate *make_edges*, and concludes with a *min_edge* operation. Assume this operation returns back edge va, corresponding to edge $vw \in E$. Assume neither va nor wa has been previously returned. As mentioned above, the Fibonacci heap $\mathcal{F}$ records edge vw as the smallest candidate for a blossom step. If vw is selected for the next step of the search algorithm, the corresponding blossom step is performed (unless vw corresponds to an augment step). Also, blossom-merging *merge* operations are executed to form the new blossom in the supporting forest. These operations place v and w in the same blossom of $\mathcal{O}$. So, wa will never be returned by future *min_edges* (by definition of that operation).

Let us estimate the extra time that these operations add to a search of Edmonds' algorithm. The algorithm for maintaining the supporting forest uses the incremental-tree set-merging algorithm of Gabow and Tarjan [20] for *merge* operations. It maintains the partition into blossoms and inner vertices in linear time. The dynamic nca algorithm of References [15, 16] use $O(n)$ time for n *add_leaf* operations and $O(m)$ time for m *nca* operations.[4] So, the extra time for the supporting forest is $O(m + n)$.

We have reduced our task to solving the *tree-blossom-merging problem*. It is defined on a tree T that is constructed incrementally by *add_leaf* operations. Wlog assume the operation $add\_leaf(x, y)$ computes $d(y)$, the depth of vertex y in its supporting tree. There are three other operations, *make_edge*, *merge*, and *min_edge*, defined as in the blossom-merging problem with the restriction that all edges of $\mathcal{E}$, i.e., the arguments to *make_edge*, are back edges.

There is no *make_blossom* operation—we assume Edmonds' algorithm does the appropriate *add_leaf* and *merge* operations. Note that $\mathcal{E}$ is a multiset: A given edge vw may be added to $\mathcal{E}$ many times (with differing costs) since although an edge $uv \in E$ will become eligible for a blossom step at most once, different u vertices can give rise to the same back edge vw. Our notation assumes the edges of $\mathcal{E}$ are directed towards the root, i.e., $vw \in \mathcal{E}$ has $d(v) > d(w)$.

As before, for any vertex x, B_x denotes the blossom currently containing x. Assume that the *merges* for a blossom step are performed in the natural bottom-up order, i.e., for $vw \in \mathcal{E}$ the search algorithm traverses the path in $\overline{\mathcal{S}}$ from B_v to B_w, repeatedly merging (the current) B_v and its parent

[4] After the conference version of this article, Cole and Hariharan [5] used a similar approach to allow these and other dynamic nca operations. In addition, their time bounds are worst-case rather than amortized.

Table 1. Edge $e = vw \in \mathcal{E}$ Has Type *l*,*s*, or *d* Defined by the Conditions Column. Merging Blossoms B_v and B_w Gives a Blossom of Rank $\geq r > r(A_u)$ (Proved in Proposition 3.2)

Type	Conditions		u	r
l	$r(e) > r(A_v)$		v	$r(e)$
s	$r(e) \leq r(A_v)$,	$r(A_v) \leq r(A_w)$	v	$\max\{r(A_v) + 1, r(A_w)\}$
d	$A_v \neq A_w$	$r(A_v) > r(A_w)$	w	$r(A_v)$

blossom. In tree-blossom-merging, we call any set resulting from a *merge* operation a "blossom" even though it need not be a blossom of Edmonds' search algorithm.

3.1 The Tree-Blossom-Merging Algorithm

This section solves the tree-blossom-merging problem in time $O(m + n \log n)$. First, it presents the basic principles for our algorithm. Then it gives the data structure, the algorithm statement, and its analysis.

Two features of the supporting forest are essentially irrelevant to our algorithm: The only role played by inner vertices, prior to becoming outer, is to contribute to depths $d(v)$. Also, the fact that supporting trees are constructed incrementally is of no consequence, our algorithm only "sees" new *merge* and *make_edge* operations.

Our strategy is to charge time to blossoms as their size doubles. We use a notion of "rank" defined for edges and blossoms: The *rank* of an edge $vw \in \mathcal{E}$ is defined by

$$r(vw) = \lfloor \log (d(v) - d(w)) \rfloor.$$

The rank of an edge is between 0 and $\lfloor \log n \rfloor$. A blossom B has *rank*

$$r(B) = \lfloor \log |B| \rfloor.$$

(Recall that in this section a blossom is a set of vertices.)

These notions are recorded in the algorithm's data structure as follows. (A complete description of the data structure is given below.) We use a simple representation of T, each vertex x in T recording its parent and depth $d(x)$. Each blossom B records its size $|B|$ and rank $r(B)$.

There are $\leq n$ *merge*s, so each can perform a constant number of time $O(\log n)$ operations, like Fibonacci tree *delete_min*s or moving $\log n$ words. There are $\leq 2m$ *make_edge*s, so each can perform $O(1)$ operations. Each vertex v always belongs to some current blossom B_v, and $\leq \log n$ *merge* operations increase the rank of B_v. So, we can charge $O(1)$ time to a vertex v every time its blossom increases in rank.

We dynamically associate each edge $vw \in \mathcal{E}$ with one of its ends, say vertex $u \in \{v, w\}$, in such a way that a future merge that places v and w in the same blossom is guaranteed to increase the rank of B_u. To do this, we assign a *type* to every edge—*long, short,* or *down*, respectively, or synonymously *l-edge, s-edge, d-edge.* (See Figure 3—the *l*, *s*, and *d* edges shown will cause a rank increase at their B end.) At any point in the execution of the algorithm every edge has a unique type. This type can change over time (we call it a reclassification).

Types are defined in Table 1 and illustrated in Figure 3. In Table 1, A_v and A_w denote the currently maximal blossoms containing v and w, respectively, when the definition is applied to classify edge vw. Edges with a "long" span, type *l*, are classified based on their span; edges with a "short" span, type *s* or *d*, are classified according to the relative sizes of the two blossoms containing their ends. Observe that the Conditions column specifies a unique type for every edge vw with $A_v \neq A_w$.

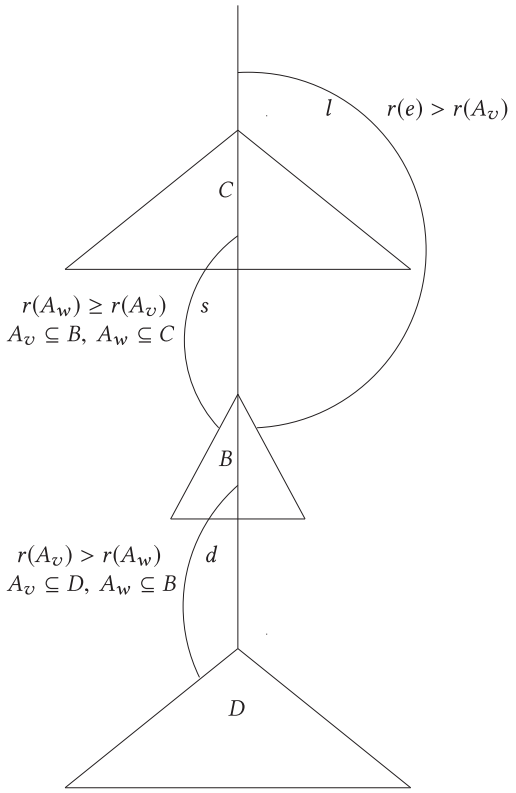

Fig. 3. l, s, and d edges associated with blossom B, i.e., $A_u \subseteq B$. l and s edges go up from B, d edges go down.

The edge type of $e = vw$ determines the end v or w that e is associated with—this is given in the u column of Table 1. The r column gives the previously mentioned guarantee – Proposition 3.2 will show that a future blossom containing both v and w has rank $\geq r$, which is greater than the current value $r(A_u)$.

As the algorithm progresses and B_v and B_w (the currently maximal blossoms containing v and w) grow, the type of e and other values in Table 1 may change. (For instance, when $B_v = B_w$ e has no type at all.) The algorithm will not track every change in e's type. Instead it examines e from time to time, each time reclassifying e according to Table 1. This is illustrated in Figure 3: The l, s, and d edges each have $u \in B$, where B is the currently maximal blossom containing u (i.e., B_u) and so $A_u \subseteq B$. Similarly, when the s edge was classified, w was in the maximal blossom A_w, which is now contained in the maximal blossom C. Ditto for the d edge, A_v, and D.

Example 1. Table 2 gives type classifications performed in a hypothetical execution of the algorithm. In particular it illustrates how edges need not get reclassified every time their status changes.

The example tracks three edges, e_1, e_2, e_3, at four points in time, the algorithm states 1 through 4. Each e_i is a parallel copy of edge vw. $r(vw) = 2$ and each classification makes its e_i an s-edge with $u = v$.

The three edges e_i are created in order of increasing i. State 1 occurs at the execution of *make_edge*(e_1). e_1 is classified in state 1 with the value $r = r(A_v) + 1 = 3$. The last column of Table 2 indicates e_1 will be reclassified in state 4 (but not before).

Later on in state 2, *make_edge*(e_2) is executed and e_2 gets classified. A_v has grown so that e_2 is an s-edge with $r = r(A_w) = 4$. The algorithm does not reclassify e_1 at this point, even though it is a copy of e_2. State 3 corresponds to *make_edge*(e_3) and, similar to e_2, e_3 is an s-edge with $r = 5$. Now all three parallel edges have different classifications.

In state 4, e_1 gets reclassified, with $r = 6$. The algorithm also examines e_2, detects it as a copy of e_1 with greater cost, and so permanently discards e_2. (A blossom step for e_1 clearly makes e_2 irrelevant.) e_3 is not reclassified at this point.

Further details of this scenario are given in Example 2. Table 3 gives similar executions for three parallel d-edges.

We call the values specified in Table 1 (specifically, the edge type, blossoms A_v and A_w, r, and u) the *type parameters* of the classification. At a given point in the algorithm, an edge $e = vw \in \mathcal{E}$ will have two pairs of blossoms that are of interest: the edge's current type parameters A_v and A_w, and the current blossoms B_v and B_w. Clearly, $A_v \subseteq B_v$ and $A_w \subseteq B_w$.

We now prove our guarantee that $r(B_u)$ has increased when v and w are in the same blossom.

Proposition 3.2. *Let $e = vw$ have type parameters A_v, A_w, r, u.*
(i) $r > r(A_u)$.
(ii) A blossom Z containing v and w has $r(Z) \geq r$.

Proof. (*i*) An l-edge has $r = r(e) > r(A_v)$ and $v = u$. An s-edge has $r \geq r(A_v) + 1$ and $v = u$. A d-edge has $r = r(A_v) > r(A_w)$ and $w = u$.

(*ii*) Suppose e is an l-edge. Z contains the path from v to w so $|Z| > d(v) - d(w) \geq 2^{r(e)} = 2^r$. Thus $r(Z) \geq r$.

If e is a d-edge, then $A_v \subseteq Z$. So, $r(Z) \geq r(A_v) = r$.

If e is an s-edge, then $A_v, A_w \subseteq Z$ and $r(A_w) \geq r(A_v)$ imply $|Z| \geq |A_v| + |A_w| \geq \max\{2^{r(A_v)+1}, 2^{r(A_w)}\}$. So $r(Z) \geq r$. □

A crucial property for the accounting scheme is that the s and d edges are limited in number. To make this precise, recall that the vertices of a blossom form a subtree of the supporting tree, so we can refer to a blossom's root vertex. (This differs from the notion of base vertex. For instance, recall that a tree-merging blossom needn't be a complete blossom of Edmonds' algorithm.) For any blossom B define $stem(B)$ to consist of the first $2|B|$ ancestors of the root of B (or as many of these ancestors that exist).

Proposition 3.3. *Let $e = vw$ where a blossom B contains v but not w and $r(e) \leq r(B)$. Then $w \in stem(B)$.*

Remarks. The proposition shows an s or d edge $e = vw$ has $w \in stem(A_v)$, since the definition of Table 1 has $r(e) \leq r(A_v)$. In fact, any blossom B containing A_v but not w has $w \in stem(B)$, since $r(A_v) \leq r(B)$ and $stem(A_v) \subseteq B \cup stem(B)$.

To illustrate, in Figure 3 $stem(D)$ contains the d-edge's w end. $stem(D)$ may contain shallower vertices, perhaps even ancestors of B or C. $stem(D)$ might contain all of $stem(B)$ or $stem(C)$.

Proof. Let $r = r(B)$. Thus $2^r \leq |B|$. Since $r(e) \leq r$, $d(v) - d(w) < 2^{r+1} \leq 2|B|$. Rearranging this to $d(v) - 2|B| < d(w)$ puts w in the stem of B. □

Let $\mathcal{M}$ be the set of all blossoms B ever formed in the algorithm that are rank-maximal, i.e., B is not properly contained in any blossom of the same rank. A given vertex belongs to at most $\log n$ rank-maximal blossoms. Thus $\Sigma_{B \in \mathcal{M}} |B| \leq n \log n$. Our plan for achieving the desired time bound involves using the proposition to charge each blossom $B \in \mathcal{M}$ $O(1)$ time for each vertex in $stem(B)$. The total of all such charges is $O(n \log n)$ and so is within our time bound.

The Data Structure. We describe the entire data structure as well as its motivation.

Each current blossom B has $\leq \log n$ lists of edges designated as $packet(B, r)$ for $r \in [r(B) + 1.. \log n]$. $packet(B, r)$ consists of edges whose classification parameters u, r have $u \in B$ and r equal to the packet index. As motivation, note that all these edges are similar in the sense of guaranteeing the same rank increase (Proposition 3.2(*ii*)). This allows the edges of a packet to be processed as a group rather than individually.[5]

Each $packet(B, r)$ is implemented as a ring, i.e., a circularly linked list. This allows two packets to be concatenated in $O(1)$ time. A header records the packet rank r and $smallest(B, r)$, the edge of smallest cost in the packet.

The packet headers of blossom B are stored in a list in arbitrary order (they are not sorted on r). Also, if $packet(B, r)$ currently contains no edges, its header is omitted. These two rules limit the space for the algorithm to $O(m)$. For graphs with $m = \Omega(n \log n)$ an alternate organization is possible, based on each current blossom having an array of $\log n$ pointers to its packets.[6]

Each current blossom B has an additional list $loose(B)$. It consists of all edges classified with $A_u = B$ but not yet belonging to a packet of B. This list is a "waiting area" for edges to be added to a packet. It allows us to omit the array of pointers to packets mentioned above. $loose(B)$ is a linked list.

The value $smallest(B)$ is maintained as the minimum cost edge in one of B's lists (i.e., a B-packet or $loose(B)$).

A Fibonacci heap $\mathcal{H}$ stores the values $smallest(B)$ for every current blossom B. It is convenient to do lazy deletions in $\mathcal{H}$: An operation $merge(A, B)$, which replaces blossoms A and B by the combined blossom C, marks the entries for A and B in $\mathcal{H}$ as deleted, and inserts a new entry for C. To do a *min_edge* operation we perform a Fibonacci heap $find\_min$ in $\mathcal{H}$. If the minimum corresponds to a deleted blossom, that entry is deleted (using *delete_min*) and the procedure is repeated. Eventually we get the smallest key for a current blossom. Its edge is returned as the *min_edge* value.

An auxiliary array $I[1..n]$ is used to perform a *gather* operation, defined as follows: Given is a collection of c objects, each one having an associated index in $[1..n]$. We wish to gather together all objects with the same index. We accomplish this in time $O(c)$, by placing objects with index i in a list associated with $I[i]$. An auxiliary list of all indices with a nonempty list allows the lists to be collected when we are done, in $O(c)$ time. Gathering operations will be done to maintain packets in *merges*.

Finally, each blossom B has several bookkeeping items: Its rank $r(B)$ is recorded. There is also a representation of the partition of $\mathcal{O}$ into blossoms. A data structure for set-merging [36] can be used: The blossom-merging operation $merge(A, B)$ executes a set-merging operation $union(A, B)$ to construct the new blossom; for any vertex v, the set-merging operation $find(v)$ gives B_v, the blossom currently containing v.

For simplicity, we will omit the obvious details associated with this bookkeeping. We can also ignore the time and space. Suppose we use a simple set-merging algorithm that does one $find$ in $O(1)$ time, all *unions* in $O(n \log n)$ time, and uses $O(n)$ space (e.g., Reference [1]). Clearly, the space and the time for *unions* are within the desired bounds for Edmonds' algorithm; the time for $finds$ can be associated with other operations. Hence we can ignore this bookkeeping.

The Algorithms. We present the algorithms for tree-blossom-merging, verify their correctness, and prove the desired time bound $O(m + n \log n)$.

[5]Our notion of packet is similar in spirit, but not detail, to the data structure of the same name in References [17, 18].

[6]Various devices can be added to this approach to achieve linear space for all m. Such an organization is used in the conference version of this article [15]. However, our current approach is more uniform and simpler.

The algorithm maintains this invariant:

(I1) The set S of all edges in a *loose*-list or packet satisfies:
  (i) every edge of S joins two distinct blossoms;
  (ii) for every two blossoms joined by an edge of $\mathcal{E}$, S contains such a joining edge of smallest cost.

(I1) guarantees that for every blossom B, $smallest(B)$ is a minimum-cost edge joining B to another blossom. In proof, (*ii*) guarantees that such a minimum-cost edge belongs to some *loose*-list or packet of B. (*i*) guarantees that this edge gives the value of $smallest(B)$ (i.e., without (*i*) it is possible that the minimum-cost edge has both ends in the same blossom and so is not useful).

As mentioned *min_edge* is a Fibonacci heap *find_min* in $\mathcal{H}$. This edge gives the next blossom step by the definition of $smallest(B)$ and invariant (I1). The total time for all *min_edges* is $O(n \log n)$, since there are $O(n)$ blossoms total and each can be deleted from $\mathcal{H}$.

make_edge Algorithm. $make\_edge(vw)$ is implemented in a lazy fashion as follows: If $B_v = B_w$ then e is discarded. Otherwise, we classify $e = vw$ as type l, s, or d, by computing $r(e), r(B_v)$ and $r(B_w)$, as well as vertex u and rank r, in time $O(1)$. e is added to $loose(B_u)$ and $smallest(B_u)$ is updated, with a possible *decrease_key* in $\mathcal{H}$. The time is $O(1)$ (amortized in the case of *decrease_key*). This time is charged to the creation of e.

merge Algorithm. An operation $merge(X, Y)$ forms a new blossom Z. (Note that a given edge e_0 that is selected by the search algorithm for the next blossom step will cause one or more such *merge* operations. The *merge* algorithm, and its analysis, does not refer to e_0.)

Let the set R_0 consist of all edges that must be reclassified because of the *merge*, i.e., the edges in packets of X or Y of rank $r \le r(Z)$, and the edges in $loose(X)$ or $loose(Y)$. The edges of R_0 are pruned to eliminate redundancies, i.e., we ensure that at most one such edge joins Z to any blossom $B \neq Z$. This is done with a gather operation using $I[1..n]$. Among all edges vw joining Z and a given blossom $B \neq Z$ (i.e., $\{find(v), find(w)\} = \{B, Z\}$) only one of smallest cost is retained. Edges with both ends in Z are discarded. These actions preserve (I1).

Let R denote the set of remaining edges. We assign each edge of R to its appropriate packet or *loose*-list, and form the final packets of Z, as follows.

Take any edge $e = vw \in R$. Compute the new type of vw using $r(e), r(B_v)$ and $r(B_w)$. If $u \notin Z$ then vw gets added to $loose(B_u)$. Specifically an s-edge vw with $w \in Z$ gets added to $loose(B_v)$. As usual, we also update $smallest(B_v)$, possibly doing *decrease_key*. Similarly, a d-edge vw with $v \in Z$ gets added to $loose(B_w)$.

The remaining edges are added to packets of Z as follows. Use the subarray $I[r(Z) + 1.. \log n]$ in a gather operation. Specifically, $I[r]$ gathers all individual edges that are r-promoters for Z and forms them into a list. It also gathers $packet(X, r)$ and $packet(Y, r)$, if they exist. These two packets are represented by their headers, not examining the individual edges, so $O(1)$ time is spent adding them to the list $I(r)$. The final $I(r)$ is taken as the list for $packet(Z, r)$.

We complete $packet(Z, r)$ by computing $smallest(Z, r)$ from $smallest(X, r)$, $smallest(Y, r)$, and the costs of all its other edges. The smallest of all these values gives $smallest(Z)$. This value is inserted into $\mathcal{H}$. The *loose* list of Z is empty.

Example 2. The algorithm achieves the states given in Table 2 as follows.

State 1: $make\_edge(e_1)$ is executed when $|B_v| = |B_w| = 4$. The algorithm classifies e_1 as an s-edge with $r = 3$. It adds e_1 to $loose(B_v)$. Next a merge makes $|B_v| = 5$. The algorithm removes e_1 from $loose(B_v)$ and examines its type parameters. e_1 is still an s-edge with $r = 3$, so it is added to $packet(B_v, 3)$.

Table 2. s-edges e_1, e_2, e_3, All Copies of vw and Their Blossom Parameters; $u = v$. e_1 and e_2 Are Reclassified in State 4

State	$r(A_v)$	$r(A_w)$	$e : r$	reclassified
1	2	2	$e_1 : 3$	4
2	2	4	$e_2 : 4$	4
3	2	5	$e_3 : 5$	-
4	4	6	$e_1 : 6$	-

Table 3. d-edges e_1, e_2, e_3, All Copies of vw, and Their Blossom Parameters; $u = w$

State	$r(A_v)$	$r(A_w)$	$e : r$	reclassified
1	3	2	$e_1 : 3$	4
2	3	2	$e_2 : 3$	4
3	5	2	$e_3 : 5$	-
4	6	3	$e_1 : 6$	-

State 2: $make\_edge(e_2)$ is executed when $|B_v| = 5 > |B_w| = 4$. The algorithm classifies e_1 as an s-edge with $r = 3$. It adds e_1 to $loose(B_v)$. Next various merges make $|B_w| = 16$, but $loose(B_v)$ does not change. Next a merge makes $|B_v| = 6$. e_2 is now an s-edge with $r = 4$, so it is transferred from $loose(B_v)$ to $packet(B_v, 4)$. The old $packet(B_v, 3)$, which contains e_1, is added to the new $packet(B_v, 3)$.

State 3: Achieved similar to state 2, with $|B_w|$ increasing to 32 and then $|B_v|$ increasing to 7. e_3 moves from a *loose* list to $packet(B_v, 5)$. e_1 and e_2 remain in the $r = 3$ and $r = 4$ packets, respectively.

State 4: Merges increase $|B_w|$ to 64. Then a merge makes $|B_v| = 16$. The edges in the rank 3 and rank 4 packets of (the old) B_v are added to R_0. e_2 is discarded by the gather operation and e_1 is added to $packet(B_v, 6)$.

Table 3 is similar. $|B_w|$ starts as 4 and increases to 5 (giving state 1), then 6 (state 2), then 7 (state 3). When $|B_w|$ increases to 8 edges e_1 and e_2 are added to R_0, e_2 is discarded and e_1 is added to the rank 6 packet (state 4).

Correctness of the Algorithm. First observe that Proposition 3.2(i) shows the new packet assigned to an edge of R actually exists. To show invariant (I1i) for Z consider the edges in packets of Z that are not examined individually, i.e., edges e in a packet of X or Y of rank $r > r(Z)$. Proposition 3.2 (ii) shows a blossom containing both ends of e has rank $\geq r > r(Z)$. Thus e satisfies (I1i). Invariant (I1ii) holds since edges are only discarded in a gather operation.

Efficiency. As mentioned the time for *make_edge* is $O(1)$. The time for *merge* is easily seen to be $O(\log n)$ (to traverse the list of packet headers) plus $O(1)$ for each edge added to R_0. The first term amounts to $O(n \log n)$ for the entire algorithm (there are $\leq n$ *merges*).

We account for the second term in two main ways. The first is to charge time to blossoms, via their stems. To do this recall that an s- or d-edge vw has $w \in stem(B_v)$ (for the current blossom B_v containing v; Proposition 3.3). We will charge the stems of blossoms $B \in \mathcal{M}$. To do this it is important to verify that that e is the only edge from B to w making the charge.

The second accounting mechanism is a system of credits. It pays for reclassifications that involve *loose*-lists. As motivation note that successive merges may move an edge $e = vw$ around a cycle of *loose*-lists and packets: For example, starting in $loose(B_w)$, e can move to a packet of B_w, to $loose(B_v)$, to a packet of B_v, and back to $loose(B_w)$. Credits will help pay for these moves. Define a credit to account for $O(1)$ units of processing time, specifically the time to process an edge $e \in R_0$. Credits are maintained according to this invariant (I2):

(I2) An edge in a *loose*-list has 1 credit if it is type s, 2 credits if it is type d.

Lemma 3.4. *The time spent on all edges of R_0 in all merges is $O(m + n \log n)$.*

Proof. We first treat l-edges. They are involved in only $O(1)$ operations that we charge to the edge itself, as follows. Observe that once an edge becomes type s or d it remains s or d ($r(e) \le r(A_v)$ has the right-hand quantity nondecreasing). So an l-edge e must start out as l in the operation $make\_edge(e)$. e starts in $loose(A_v)$. The first merge involving A_v makes e type s, d, or l. In the last case e is added to a rank $r(e)$ packet. It remains in this packet until a merge makes $r(B_v) \ge r(e)$. That changes e to type s or d. So e uses total time $O(1)$ as an l-edge, which is charged to e.

Now consider an s or d edge $e = vw$. $make\_edge(e)$ adds e to the appropriate *loose* list with credits corresponding to (I2).

Suppose $e \in R_0$ in an operation $merge(X, Y)$ that forms the new blossom Z. If $e \notin R$ it pays for $O(1)$ time spent processing e. e is then discarded, so it is completely accounted for.

Suppose $e \in R$. Let e start out with classification parameters A_v, A_w, u, r. For definiteness, let X be the blossom containing an end v or w of e. We examine the two possible starting types for e.

Case *e starts as an* s*-edge*: There are two starting possibilities:

Subcase *e starts in a packet*: e starts in $packet(X, r)$. This implies $X \in \mathcal{M}$.[7] In proof, the definition of X's packets shows $r(X) < r$. Since the merge has $e \in R_0$, $r \le r(Z)$. Combining gives $r(X) < r(Z)$. Thus X is rank-maximal.

e is the unique edge of R that is directed to w. We charge $w \in stem(X)$ at most three credits: one for processing e and two more if e becomes a d-edge. Now (I2) holds for e.

Clearly $X \in \mathcal{M}$ implies this is the only time the algorithm charges $w \in stem(X)$.

Subcase *e starts in $loose(X)$*: (I2) shows it has one credit. If e remains an s-edge it gets added to a packet. The credit pays for processing e.

Suppose e becomes a d-edge, so it gets added to $loose(B_w)$. Since e changes from s to d the definitions show $r(A_v) \le r(A_w) \le r(B_w) < r(B_v)$. Note $A_v = X, B_v = Z$. (The former holds since, in general, an edge in a list $loose(B)$ is in R_0 in the first merge involving B.) So $r(X) < r(Z)$. Thus $X \in \mathcal{M}$. So, we can charge $w \in Stem(X)$ as in the previous subcase.

Case *e starts as a d-edge*: Again there are two possibilities.

Subcase *e starts in a loose-list*: e requires one credit for processing, and one more if it becomes an s-edge. It starts with two credits (I2) so it can pay for the reclassification.

Subcase *e starts in a packet*: We wish to charge the reclassification to a stem containing w. Let A_v^+ be the rank $r = r(A_v)$ blossom in $\mathcal{M}$ that contains A_v. (A_v need not be rank-maximal. It may even be that A_v is still the currently maximal blossom containing v and $A_v \subset A_v^+$.) We claim $w \in stem(A_v^+)$. To prove this note that $A_v \subseteq A_v^+$ implies we need only show $w \notin A_v^+$. A blossom B containing v and w contains A_v and Z. $e \in R_0$ implies $r = r(A_v) \le r(Z)$. Thus $r(B) \ge r + 1 > r(A_v^+)$. This guarantees $w \notin A_v^+$.

[7] This holds even if e is a d-edge, but we do not use this fact.

We charge the reclassification of e to $w \in stem(A_v^+)$. The charge is $O(1)$ time to account for the processing of e in this merge. In addition we charge one credit if e becomes an s-edge.

The gather step ensures w is charged only once in this merge. Furthermore, no future merge will charge $w \in stem(A_v^+)$. In proof, note that the reclassification has $r(B_w) = r(Z) \geq r = r(A_v)$. Thus, future blossoms containing w also have rank $\geq r$, and these blossoms will not have a packet of rank $r = r(A_v^+)$. So, this is the only charge the algorithm makes to $w \in stem(A_v^+)$. □

Example 3. We add some details to the description of Table 3 in Example 2, by tracking the value of the ordered pair $(|B_v|, |B_w|)$ as the algorithm progresses. It starts as $(8, 4)$ and advances to $(8, 5)$ in state 1, $(9, 6)$ in state 2, $(32, 7)$ in state 3, and $(64, 8)$ in state 4. So, when e_1 gets reclassified in state 4, it starts with $|A_v| = 8$. A_v is not rank-maximal, A_v^+ is the state 2 blossom that has size $|A_v^+| = 9$.

As another example, consider an edge $e = vw$ that is added to $packet(B_w, 3)$ as a d-edge with classification parameters $r(A_v) = 3, r(A_w) = 2$. A merge makes $r(B_w) = 3$ and e is added to $loose(B_v)$. A merge enlarges B_v. For definiteness, call B the resulting blossom, and assume its rank remains 3 in the merge. So e is added to $packet(B, 4)$ as an s-edge. Another merge makes $r(B_v) = 4$, so e moves to $loose(B_w)$.

The last merge makes $B \in \mathcal{M}$. Thus, the move of e to $loose(B_v)$ is charged to $w \in stem(B)$. The move of e to $loose(B_w)$ is also charged to $w \in stem(B)$. So, this example shows a given stem entry can be charged two times.

Having analyzed the algorithms for *min_edge*, *make_edge*, and *merge*, we conclude that our tree-blossom-merging algorithm achieves the desired time bound:

Theorem 3.5. *The tree-blossom-merging problem can be solved in time* $O(m + n \log n)$.

4 b-MATCHING

This section presents a simple generalization of Edmonds' algorithm to b-matching. The major complication is that b-matchings allow two varieties of blossoms, which we call "light" (analogous to ordinary matching) and "heavy." Our goal is an algorithm that is as similar to ordinary matching as possible. This involves minimizing the role of heavy blossoms. Heavy blossoms seem impossible to avoid but our algorithm keeps them "hidden."

Section 4.1 gives the basic properties of b-matching blossoms. Section 4.2 presents the generalized algorithm. Section 4.3 gives the analysis, eventually showing we find a maximum b-matching in time $O(b(V)(m + n \log n))$. Section 4.4 extends the algorithm to achieve the strongly polynomial time bound $O((n \log n)\ (m + n \log n))$. Both of these bounds match the best-known time bounds for bipartite graphs.

A degree-constraint function b assigns a nonnegative integer to each vertex. We view b-matching as being defined on a multigraph. Every edge has an unlimited number of copies. In the context of a given b-matching, an edge of the given graph has unlimited number of unmatched copies; the number of matched copies is specified by the b-matching. In a *partial b-matching* every vertex v has degree $\leq b(v)$. In a *(perfect) b-matching* every vertex v has degree exactly $b(v)$. Note that "b-matching" (unmodified) refers to a perfect b-matching, our main concern.

We use these multigraph conventions: Loops are allowed. A cycle is a connected subgraph that is regular of degree 2. It can be a loop, 2 parallel edges, or an undirected graph cycle. A *trail* is a path that is allowed to repeat vertices but not edges.

Contracting a subgraph does not add a loop at the contracted vertex (all internal edges including loops disappear). We will even contract subgraphs that contain only a loop. We use the following notation for contractions: Let $\overline{G}$ be a graph derived from G by repeatedly contracting an arbitrary

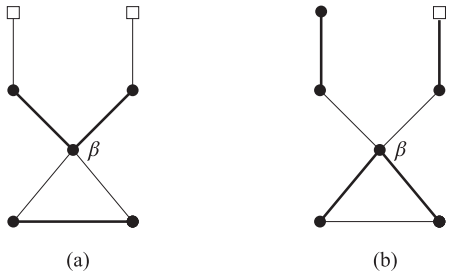

Fig. 4. An augmenting trail (a) and the augmented matching (b).

subgraph. (The subgraph may even contain previously contracted vertices.) V ($\overline{V}$) denotes the vertex set of G ($\overline{G}$), respectively. A vertex of $\overline{G}$ that belongs to V (i.e., it is not in a contracted subgraph) is an *atom*. We identify an edge of $\overline{G}$ with its corresponding edge in G. Thus an edge of $\overline{G}$ incident to a contracted vertex is denoted as xy, where x and y are V-vertices in distinct $\overline{V}$-vertices, and $xy \in E$. If H is a subgraph of $\overline{G}$, the *preimage* of H is a subgraph of G consisting of the edges of H, plus the preimages of subgraphs whose contractions are vertices of H, plus the atoms of H. $\overline{V}(H)$ ($V(H)$) denotes the vertex set of H (the preimage of H), respectively. Similarly, $\overline{E}(H)$ ($E(H)$) denotes the edge set of H (the preimage of H), respectively.

4.1 Blossoms

This section presents the basic properties of b-matching blossoms. We define blossoms in two steps. First we specify their topology. This includes showing how blossoms are updated when the matching gets augmented. Then we specialize the definition to "mature" blossoms, those that can have positive dual variables.

Unlike ordinary matching, contracted b-matching blossoms do not behave exactly like original vertices. For instance, Figure 4(a) shows an augmenting trail—interchanging matched and unmatched edges along this (nonsimple) trail enlarges the matching to Figure 4(b). (As in all figures square vertices are free and heavy edges are matched. In Figure 4(b), one vertex remains free after the augment—its b-value has not been achieved.) The triangle is a b-matching blossom (just like ordinary matching). In Figure 4(a), contracting this blossom gives a graph of five vertices that has no augmenting path. Contracted blossoms behave in a more general way than ordinary vertices.

When a blossom becomes "mature," it behaves just like a vertex—in fact, a vertex with b-value 1 just like ordinary matching! It also behaves like an ordinary matching blossom in that its z-value can be positive (in contrast an immature blossom, e.g., the blossom of Figure 4, cannot have positive z).

We will define a blossom in terms of its topology—it is a subgraph that when contracted can behave like a vertex or like Figure 4. We will then specialize this notion to the case of mature blossoms. For completeness, we give two versions of the topological definition. Definition 4.1 is simpler. It defines a blossom as a type of ear decomposition. Definition 4.2 is more useful algorithmically.

Let G be a graph with a partial b-matching, i.e., every vertex v is on $\leq b(v)$ matched edges. A trail is *closed* if it starts and ends at the same vertex. A trail is *alternating* if for every two consecutive edges exactly one is matched. The first and last edges of a closed trail are not consecutive. (For instance, a loop is an alternating closed trail, whether it is matched or not.) Say that the *M-type* of an edge is M or $E - M$, according to the set that contains it. The following definition is illustrated in Figure 5.

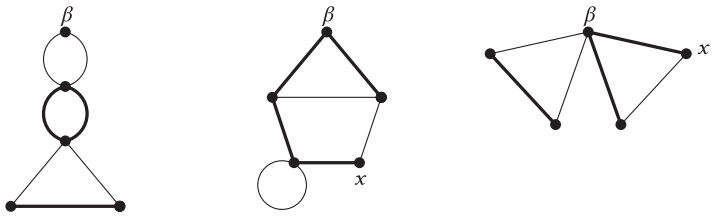

Fig. 5. Blossoms with base vertex β. (a) is light, (b) is heavy, (c) is heavy or light.

Definition 4.1 (Ear Blossom). A *blossom* is a subgraph of G recursively defined by two rules:

(a) A closed alternating trail of G is a blossom if it starts and ends with edges of the same M-type.
(b) In a graph derived from G by contracting a blossom α, let C be a closed alternating trail that starts and ends at α but has no other occurrence of α. The preimage of C in G is a blossom.

An example of (a) is a loop.

For uniformity, we extend the notation of (b) to (a): C denotes the blossom's trail and α denotes its first and last vertex. The *base vertex* of a blossom is α in (a) and the base vertex of α in (b). Clearly, the base is always in V. The base vertex of blossom B is denoted $\beta(B)$ (or β if B is clear). The *M-type* of B is the M-type of the starting and ending edges in (a), and the M-type of α in (b). A blossom of M-type M is called *heavy*, and M-type $E - M$ is *light*.

Example 1. In Figure 5(a), the entire graph forms a closed alternating trail that starts and ends at β. It is a blossom with base vertex β, by part (a) of the definition. There is essentially no other way to view the graph as a blossom, since the two edges at β do not alternate.

In Figure 5(b), the unique closed alternating trail starting and ending at β is the triangle. So it is a minimal blossom. Contracting it to α gives a graph with a unique closed alternating trail that starts and ends at α. So the graph is a blossom with base vertex β. Again, this is essentially the only way to parse this graph as a blossom.

Figure 5(c) is a light blossom with base β if we start the decomposition with the left triangle. Starting with the right triangle gives a heavy blossom based at β. No other blossom decomposition is possible.

Only light blossoms occur in ordinary matching, and they are the main type in b-matching. We note two instances of the definition that will not be treated as blossoms in the algorithm. Both instances are for a light blossom B. If $d(\beta, M) \le b(\beta) - 2$, then (a) actually gives an augmenting trail. Secondly the definition allows $d(\beta, \gamma(V(B), M)) = b(\beta)$. This never holds in the algorithm – β is either on an edge of $\delta(V(B), M)$ or $d(\beta, M) < b(\beta)$.

Consider a blossom B with base vertex β. Similar to ordinary matching, each vertex $v \in V(B)$ has two associated $v\beta$-trails in $E(B)$, $P_0(v, \beta)$ and $P_1(v, \beta)$, with even and odd lengths, respectively. Both trails are alternating and both end with an edge whose M-type is that of B (unless the trail has no edges). The starting edge for $P_1(v, \beta)$ has the same M-type as B; it has the opposite M-type for $P_0(v, \beta)$. As examples, $P_1(\beta, \beta)$ is the entire trail in Definition 4.1(a). This trail could be a loop. $P_0(\beta, \beta)$ is always the trivial trail (β). It is the only trivial P_i trail.

The recursive definitions of the P_i trails follow easily from Definition 4.1. We omit them since they are a special case of the P_i trails defined below.

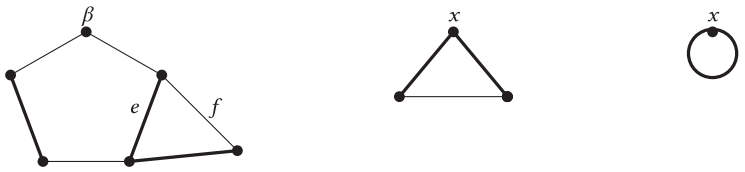

Fig. 6. More blossom examples: (a) is a blossom with base β. Adding (b) or (c) to Figure 5(b) gives a larger blossom.

The second definition of blossom mimics the structures discovered in the algorithm. As in ordinary matching the algorithm contracts blossoms as they are discovered. So the definition works in graphs $\overline{G}$ whose vertices are either atoms or blossoms.

Definition 4.2 (Algorithmic Blossom). Let $\overline{G}$ be a graph derived from G by contracting a family $\mathcal{A}$ of zero or more vertex-disjoint blossoms. Let C be a closed trail in $\overline{G}$ that starts and ends at a vertex $\alpha \in \overline{V}(G)$. The preimage of C is a *blossom* B with *base vertex* $\beta(B)$ if C has the following properties:

If α is an atom, then C starts and ends with edges of the same M-type. B has this M-type and $\beta(B) = \alpha$.

If $\alpha \in \mathcal{A}$ then B has the same M-type as α and $\beta(B) = \beta(\alpha)$.

If v is an atom of C, then every two consecutive edges of $\delta(v, C)$ alternate.

If $v \in \mathcal{A} \cap C$, then $d(v, C) = 2$. Furthermore, if $v \neq \alpha$ then $\delta(\beta(v), C)$ contains an edge of opposite M-type from v.

As before, we abbreviate $\beta(B)$ to β when possible. Also, B is *heavy* (*light*) if its M-type is M ($E - M$), respectively.

Example 2. The graph of Figure 5(a) can be parsed as a blossom by starting with the triangle (a light blossom), enlarging it with the two incident edges (heavy blossom), and enlarging that with its two incident edges (light blossom). An advantage over Definition 4.1 is that each of these blossoms is a cycle rather than a closed trail. The algorithm will use this property.

Figure 6(a) is a blossom. It can be decomposed starting with the five-cycle or starting with the triangle. If we replace edge e by edge f in the matching, the triangle remains a blossom but the overall graph does not.

Suppose the graph of Figure 5(b) is enlarged by adding the triangle of Figure 6(b) at the vertex x. The graph is a blossom. A decomposition can start by contracting the triangle. Alternatively it can delay the triangle contraction until the end. If we use Definition 4.1, we must delay the triangle contraction until the end.

Suppose instead that Figure 5(b) is enlarged by adding the loop of Figure 6(c) at x. The graph remains a blossom using Definition 4.2, since we can start by contracting the loop. This is the only possibility—if we start by contracting Figure 5(b) as in Example 1, the loop disappears in the final contraction, so it is not part of the blossom. So, this graph is not a blossom using Definition 4.1.

When all b-values are 1, the problem is ordinary matching, and Definition 4.2 is equivalent to ordinary matching blossoms if we change "closed trail" to "closed path."[8] We will show the

[8]Without the change, a blossom can contain the matched edge incident to its base β, e.g., it has the form $e_1, \ldots, e_2, e_3, \ldots, e_k$, where each e_i is incident to β and e_3 is matched.

two definitions of blossom are essentially equivalent. The main difference is that they need not provide the same edge sets; Figure 6(c) gives the simplest of examples. Instead we show they provide blossoms with the same vertex sets $V(B)$. Strictly speaking, the lemma is not needed in our development since our algorithm only uses algorithmic blossoms.

Say two blossoms are *equivalent* if they have the same M-type, base vertex, and vertex set (the latter meaning $V(B) = V(B')$). For instance, the blossoms of Figure 5(b) and its enlargement with Figure 6(c) are equivalent.

Let $\mathcal{A}^*$ denote the family of blossoms involved in the recursive construction of B, i.e., $\mathcal{A}^*$ consists of $\mathcal{A}$ plus the $\mathcal{A}^*$ family of every blossom $A \in \mathcal{A}$. Define $\mu(B) = |\mathcal{A}^* - \alpha^*|$, the total number of steps in decompositions for all the blossoms of $\mathcal{A} - \alpha$. The next proof will induct on this quantity.

Lemma 4.3. *The ear blossoms and algorithmic blossoms are equivalent families. More precisely every ear blossom is an algorithmic blossom. Every algorithmic blossom has an equivalent ear blossom.*

Proof. The first assertion is obvious. To prove the the second assertion, consider an algorithmic blossom B. Let $\mathcal{A}, C$, and α be as in the definition. We prove that B has an equivalent ear blossom by induction on $\mu(B)$. The base case $\mu(B) = 0$ corresponds directly to Definition 4.1.

Take any algorithmic blossom $A \in \mathcal{A} - \alpha$. Let e be an edge of $\delta(\beta(A), C)$ with opposite M-type from A, let f be the other edge of $\delta(A, C)$, and let v be the end of f in $V(A)$. (If there are two possibilities for e, choose arbitrarily.) Define a vertex $\overline{v} \in C(A)$ as follows: If v is in a contracted blossom of $C(A)$ then $\overline{v}$ is that blossom. Otherwise, (vertex v itself is in $C(A)$) $\overline{v}$ is an arbitrarily chosen occurence of v in $C(A)$,

Case $\overline{v} = \alpha(A)$: Note in the two subcases when $v = \overline{v}$, this implies $v = \beta(A)$.

Subcase $v \neq \overline{v}$ *or* $v = \overline{v}$ *and* e *and* f *alternate*: In C, replace A by $\alpha(A)$. In both cases, this gives an algorithmic blossom B_1. Since $\mu(B_1) < \mu(B)$ it has an equivalent ear blossom E_1. Contract it to $\overline{E}_1$. The closed trail consisting of $\overline{E}_1$ and $C(A)$ is an algorithmic blossom with μ-value $< \mu(B)$. (This motivates the definition of μ: Inducting on $|\mathcal{A}^*|$ can fail here.) By induction, it has an equivalent ear blossom E_2. E_2 is the desired ear blossom equivalent to B.

Subcase $v = \overline{v}$ *and* e *and* f *have the same M-type*: Let A_1 be the minimal blossom of A^* that has base $\beta(A) = v$. In C replace the contracted blossom A by $C(A_1)$. This gives an algorithmic blossom B_1 with $\mu(B_1) < \mu(B)$ (even if $A_1 = A$). Induction shows B_1 has an equivalent ear blossom. If $A_1 = A$, we are done. Otherwise, contract B_1 to a vertex E_1. E_1 with the blossoms of $A^* - A_1$ is an algorithmic blossom. By induction it has an equivalent ear blossom. This is the desired ear blossom equivalent to B.

Case $\overline{v} \neq \alpha(A)$: Choose an edge $g \in \delta(\overline{v}, C(A))$ as follows: If $v \neq \overline{v}$, i.e., $\overline{v}$ is a contracted blossom, g is an edge of $\delta(\beta(\overline{v}))$ of opposite M-type from $\overline{v}$. If $v = \overline{v}$ then g is an edge of $\delta(v)$ of opposite M-type from f.

Let P be the $\overline{v}\alpha(A)$-subtrail of $C(A)$ that starts with edge g. Let $Q = C(A) - P$ be the other $\overline{v}\alpha(A)$-subtrail of $C(A)$.

Replace the contraction of A by P in C. This alternating trail forms an algorithmic blossom B_1. Since $\mu(B_1) < \mu(B)$ it has an equivalent ear blossom E_1. Contract E_1 to a vertex $\overline{E}_1$.

We will form an algorithmic blossom equivalent to B by adding Q as follows. Let Q_1 be the subtrail of Q starting with its first edge (which is incident to E_1) and ending with the first edge that enters E_1. Q_1, which starts and ends with vertex $\overline{E}_1$, is the closed trail of an algorithmic blossom. It has an equivalent ear blossom E_2. If $V(Q - Q_1) \subseteq V(E_2)$ then E_2 is the desired ear blossom equivalent to B. (Note this can hold even when $Q_1 \neq Q$, i.e., edges of Q that follow Q_1 remain in vertices of Q_1.) Otherwise, continue in the same manner, defining Q_2 as the subtrail of $Q - Q_1$ starting with

the first edge that leaves E_2 and ending with the first edge that enters it. Eventually Q is exhausted and we get the desired ear blossom. □

From now on, we only use algorithmic blossoms. The next step is to show that the P_i trails defined previously exist in algorithmic blossoms.

Similar to ordinary matching, we define $P_0(\beta(B), \beta(B))$ to be the trail of no edges, $(\beta(B))$. (In fact, this is the only possibility in blossoms where all edges incident to $\beta(B)$ have the same M-type.) It is easy to see that any other trail $P_i(v, \beta(B))$ must contain at least one edge. This makes $P_0(\beta, \beta)$ slightly different from all other P_i's, we will point out how it is always consistent with the general case.

Take any trail $P_i(v, \beta(B))$ of positive length.

$$\text{The first edge of } P_i(v, \beta(B)) \text{ has the same M-type as } B \iff i = 1. \tag{4.1}$$

Equation (4.1) holds for $P_0(\beta, \beta)$ by convention.

Lemma 4.4. *Trails $P_i(v, \beta)$, $i = 0, 1$ always exist (in an algorithmic blossom).*

Proof. The overall argument is by induction on $|V(B)|$. Let $\overline{G}, \mathcal{A}, C, \alpha, \beta$ all refer to Definition 4.2 for B. Consider two possibilities for v.

Case *v is an atom in $\overline{G}$*: We first specify $\overline{P}$, the image of the desired trail $P_i(v, \beta)$ in $\overline{G}$. Then we enlarge $\overline{P}$ to $P_i(v, \beta)$ by specifying how it traverses the various blossoms of $\mathcal{A}$.

If $v = \alpha$, there is nothing to prove for $i = 0$ ($P_0(v, \beta) = (\beta)$). If $i = 1$ then $\overline{P}$ is the entire trail C.

If $v \neq \alpha$, choose an arbitrary occurrence of v in C. The edges of C can be partitioned into two $v\alpha$-trails in $\overline{G}$, one starting with a matched edge, the other unmatched. Choose $\overline{P}$ from among these trails so its starting edge has the M-type specified by Equation (4.1).

To convert $\overline{P}$ to the desired trail, for every contracted blossom $\overline{A}$ in $\overline{P}$ we enlarge $\overline{P}$ by adding a trail Q that traverses $V(A)$ correctly. To do this, let e and f be the edges of $\overline{P}$ incident to $\overline{A}$, with $e \in \delta(\beta(A))$ of opposite M-type from A. (If there are two possibilities for e choose arbitrarily. If $\overline{A} = \alpha$, then e does not exist but f does, since $v \notin \alpha$.) Let u be the end of f in $V(A)$. Let Q be the trail $P_j(u, \beta(A))$ of blossom A, with j chosen so the trail alternates with f at u. (Using Equation (4.1), this means f has opposite M-type from A iff $j = 1$. For example, suppose $u = \beta(A)$. If f has M-type opposite from A, then $Q = P_1(\beta(A), \beta(A))$. If f has the same M-type as A, then $Q = P_0(\beta(A), \beta(A)) = (\beta(A))$.)

By definition, Q alternates with e at its other end $\beta(A)$ (even when $Q = (\beta(A))$). So Q (or its reverse) is the desired trail traversing $V(A)$. Enlarge $\overline{P}$ with every such Q. Each $\overline{A}$ occurs only once in B, so no edge is added to $\overline{P}$ more than once, i.e., the final enlarged $\overline{P}$ is a trail.

Case *v is a vertex of a blossom $A \in \mathcal{A}$*: If $A = \alpha$, then $P_i(v, \beta)$ for blossom B is that trail as defined for blossom A.

If $A \neq \alpha$, we construct the desired trail as the concatenation of two trails,

$$P_i(v, \beta) = P_j(v, \beta(A))\, P_k(\beta(A), \beta).$$

We first specify the values of j and k. The M-type of the first edge of P_j is determined by Equation (4.1), applied to the given values i, v, β, and B. This M-type determines the value of j by again applying Equation (4.1), to the M-type and the given $v, \beta(A)$, and A.

To make the concatenated trail alternating, P_k starts with the edge of $\delta(\beta(A), C)$ of opposite M-type from A. So, the value of k is determined by applying Equation (4.1), to the M-type opposite from A and the given $\beta(A), \beta$, and B.

The trail $P_j(v, \beta(A))$ exists by induction. The trail $P_k(\beta(A), \beta)$ exists by applying the previous case (v atomic) to vertex $\beta(A)$. □

Paths P_i are not simple, in general. However, it is easy to see the above proof implies any trail $P_i(v, \beta)$ passes through any $A \in \mathcal{A}^*$ at most once, and if so it traverses some trail $P_j(v, \beta(A))$, $v \in V(A)$ (possibly in the reverse direction).

Example 3. In Figure 5(c), consider $P_1(x, \beta)$. The length of this trail is one if the blossom is heavy, five if the blossom is light.

As previously mentioned, the blossoms B in our algorithm have a bit more structure than Definition 4.2. (Most of the following properties are the same as ordinary matching.) A light blossom will always have $d(\beta, \gamma(V(B), M)) < b(\beta)$. Furthermore, $C = C(B)$ will always be a cycle in $\overline{G}$. Thus P_i-trails have a path-like structure. More precisely for any $v \in V(B)$, $P_i(v, \beta) \cap C$ is a path in $\overline{G}$ (i.e., no repeated vertex) with one exception: When α is atomic (so $\alpha = \beta$), $P_1(\beta, \beta) \cap C$ repeats vertex β. Repeated vertices present a difficulty for expanding a blossom—an expanded blossom must be replaced in the search forest $\overline{\mathcal{S}}$ by a path, not a trail. The special structure of blossoms allows this to be done, as shown in Section 4.2.

Augmenting Trails

The algorithm enlarges a b-matching using a generalization of augmenting paths in ordinary matching. Throughout this subsection, let M be a partial b-matching. An *augmenting trail* for M is an alternating trail AT such that $M \oplus AT$ is a b-matching with one more edge than M. To *augment* M, we replace it by $M \oplus AT$.

Our algorithm finds augmenting trails in blossoms as follows. A vertex $v \in V(G)$ is *free* if $d(v, M) \le b(v) - 1$. Consider a multiset $\{v, v'\}$ of two free vertices, where $v = v'$ only if $d(v, M) \le b(v) - 2$. We will define an *augmenting blossom* AB for v, v' and a corresponding augmenting vv'-trail AT.

Create an artificial vertex ε, with an artificial matched edge to each of v, v'. ($v = v'$ implies the two matched edges are parallel.) AB is a blossom with base vertex ε, wherein each of v, v' is either an atom or the base of a light blossom in $\mathcal{A}(AB)$. AT is $P_0(v, \varepsilon)$ with its last edge (v', ε) deleted. Clearly AT is an alternating vv'-trail. The two possibilities for each end v, v' both ensure the first and last edges of AT are unmatched, i.e., $M \oplus AT$ is a valid partial b-matching with greater cardinality than M.

This definition allows an end v with $d(v, M) \le b(v) - 2$ to be the base of a light blossom B that is properly contained in the augmenting trail. But B itself suffices. Our algorithm will always use B, not a larger blossom, to form AB.

The algorithm's *augment step* starts with an augmenting blossom and rematches the trail $P_0(v, \varepsilon)$ as above.

In Figure 7, an augment rematches the trail $P_0(v, \beta)$ through blossom B. Blossom A changes from light to heavy while B remains light. The next lemma shows that, in general, blossoms are preserved by the rematching of an augment step.

Lemma 4.5. *Consider a graph with partial b-matching M. Let B be a blossom (not necessarily maximal) with base vertex β. Suppose a trail $P_i(v, \beta)$ through B ($v \in V(B)$) is rematched. The new matching $M' = M \oplus P_i(v, \beta)$ has a blossom B' with the same subgraph as B and base vertex $\beta(B') = v$. The M-type of B' is that of the first edge of $P_i(v, \beta)$ in M', unless $M' = M$.*

Remarks. $M' = M$ only for the trail $P_0(\beta, \beta)$. Obviously, B' is identical to B in this case.

The lemma implies B and B' have the same M-type for $i = 0$ (since $P_0(v, \beta)$ begins with an edge of opposite M-type from B) and opposite M-types for $i = 1$.

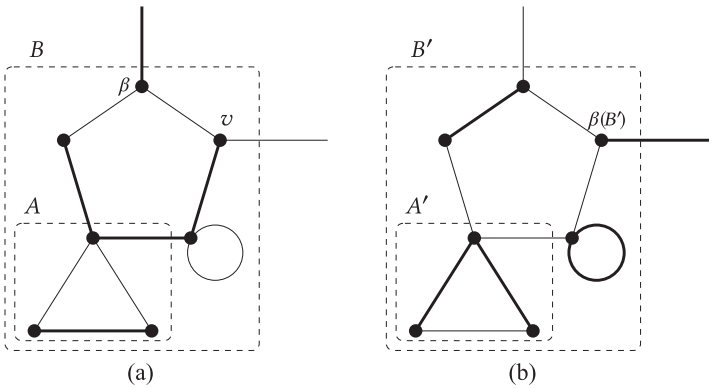

Fig. 7. Rematching a trail.

Proof. By induction, assume the lemma holds for blossoms smaller than B.

We use the following notation: $C(B)$ contains its end vertex α and a vertex $\overline{v}$ that is either the atom v or a blossom containing v. We will show the new blossom B' has $\alpha(B') = \overline{v}$. So α is either identical to $\overline{v}$ or it is an interior vertex of $C(B')$. Furthermore, the vertices of $C(B) - \{\alpha, \overline{v}\}$ remain interior in $C(B')$.

For the main argument, consider a vertex $x \in V(\overline{G})$ on $C(B)$. Assume x is on the portion of $C(B)$ that gets rematched or else there is nothing to prove. We will show that in M', x satisfies all the relevant conditions in Definition 4.2 for B', as well as the relevant conditions of the lemma.

First, suppose x is an atom. Consider a fixed occurrence of x in $C(B)$. This occurrence is either an interior vertex of $C(B)$ or, as the first/last vertex of $C(B)$, it is the base vertex β of $C(B)$.

Case $x = \beta$: Let $e, f \in \delta(x)$ be the first and last edges of $C(B)$.

Subcase $x = v$: If $P_0(\beta, \beta)$ gets rematched, then nothing changes in B. If $P_1(\beta, \beta)$ gets rematched, then obviously this creates a blossom B'. B' has the M-type of the rematched e and f. In both cases, $x = \beta(B')$ and all claims of the lemma hold.

Subcase $x \neq v$: e and f have the same M-type in M. Since $\beta = x \neq v$, $P_i(v, \beta)$ contains exactly one of e, f. So, these edges alternate in M'.

Case $x \neq \beta$: Let $e, f \in \delta(x)$ be the two edges of $C(B)$ corresponding to this occurrence of x. e and f alternate in M.

If $x \neq v$, then both edges are in $P_i(v, \beta)$ and so they alternate in M'.

The remaining case is $x = v$. Exactly one edge of the pair e, f is on $P_i(v, \beta)$, say e. So e gets rematched but f does not. Thus e and f have the same M-type in M', x is the base of the blossom B', and its M-type is that of the rematched e.

Now assume x is a contracted blossom. The inductive assumption shows x is a blossom in M', call it x'.

Case $x \neq \overline{v}$: (x may or may not be α in this case.) Let $P_i(v, \beta)$ traverse x along the trail $P_j(u, \beta(x))$. So, there is an edge $f \in C(B) \cap P_i(v, \beta)$ ending at u.

First, suppose $P_j(u, \beta(x))$ has at least one edge of x. Its first edge e has opposite M-type from f. The inductive assertion shows x' has M-type that of the rematched e, which is opposite that of the rematched f. Since f is incident to $u = \beta(x')$, f is the edge of $\delta(\beta(x'), C(x'))$ required for blossom B' in Definition 4.2. We conclude Definition 4.2 is satisfied for x'.

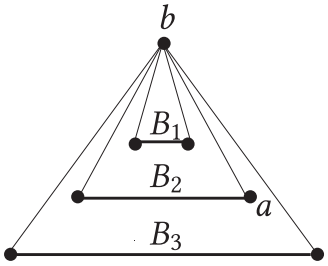

Fig. 8. $V(B_1) \subset V(B_2) \subset V(B_3)$ and $\beta(B_i) = b$. Rematching $P(a, b)$ makes $\beta(B_2) = \beta(B_3) = a \neq b = \beta(B_1)$. Subsequently rematching $P(b, a)$ restores all bases to b.

In the remaining case, no edge of subgraph x is rematched, i.e., f is incident to $u = \beta(x')$ and has the same M-type as x, so $P_j(u, \beta(x)) = (\beta(x'))$. The rematched f has opposite M-type from $x = x'$. As before, Definition 4.2 is satisfied for x'.

Case $x = \overline{v}$: By induction $v = \beta(x')$, so we take $v = \beta(B')$ and $x' = \alpha(B')$. B' has the same M-type as x'. Let e and f be the two edges of $C(B)$ incident to x. In B' there is no constraint on the M-types of e and f.

Suppose x contains an edge of $P_i(v, \beta)$. The inductive assertion for x shows x' and B' have the M-type of the rematched first edge of $P_i(v, \beta)$, as required by the lemma.

Suppose x does not contain an edge of $P_i(v, \beta)$. Then $v = \beta(x)$ and $P_i(v, \beta)$ uses the trail $P_0(v, v)$ through x. If $x \neq \alpha$ then before rematching, the first edge of $P_i(v, \beta)$ has opposite M-type from x. So, after rematching, this edge has the same M-type as $x = x'$ and B', as claimed in the lemma. If $x = \alpha$ then no edge of B is rematched, and again the lemma holds. □

Figure 8 illustrates how rematching can change blossom bases: Blossoms with a common base can have different bases after rematching, and vice versa.

Mature Blossoms

We turn to the completeness property for blossoms. As in the algorithm for ordinary matching, linear programming z-duals will be positive only on blossoms. However, complementary slackness requires that a blossom B with $z(B) > 0$ is completely matched and has exactly one incident matched edge,

$$b(V(B)) = 2|\gamma(V(B), M)| + 1. \tag{4.2}$$

(This principle is reviewed in Appendix B.) If B is light, this matched edge will be incident to $\beta(B)$. (This is required by Definition 4.2 in the case that B is not maximal.) So light blossoms are similar to ordinary matching blossoms.

If B is heavy it may have exactly one incident matched edge. But this is irrelevant in our algorithm—heavy blossoms will not be assigned positive z-values. In detail, heavy blossoms created in a search are immediately absorbed into a light blossom. The reader may look ahead to Figure 11(d)–(f) for a simple example. It is based on the fact that unmatched edges have an unlimited number of copies in b-matching. Heavy blossoms can be created in an augment, as in Figure 7, but our algorithm "hides" them, as shown below.

This discussion motivates the following definition:

For any vertex v with $b(v) \geq 1$ define the function b_v by decreasing $b(v)$ by 1, keeping all other b-values the same. A blossom based at β is *mature* if $\gamma(V(B), M)$ is a (perfect) b_β-matching.

Our search algorithm will extend a blossom to make it mature before any dual adjustment makes its z-value positive (see Lemma 4.11).

Now consider the transition from one search to the next. First consider an augment step. Blossoms with positive dual values must remain mature. This is guaranteed by the next lemma, illustrated by B in Figure 7 assuming it is mature.

Lemma 4.6. *Let B be a blossom, not necessarily maximal. If B starts out light and mature and gets rematched in an augment, it remains light and mature.*

Proof. Let AB be the augmenting blossom. B and AB have different base vertices ($\beta(AB)$ is artificial). Thus AB contains exactly two edges incident to B at least one of which is incident to $\beta(B)$ and is matched. (This follows from Definition 4.2. Note the matched edge may be artificial.) Let f be the other edge. Since B is mature f is unmatched. The augment rematches f and the P_i trail through B joining f to $\beta(B)$. Thus, Lemma 4.5 shows the rematched blossom is light and mature. □

Now consider the contracted graph $\overline{G}$ immediately after the matching is augmented. Some maximal blossoms may be immature (discovery of the augmenting trail prevents these blossoms from growing to maturity). Such immature blossoms B should be discarded. (For instance, a contracted blossom B incident to >1 matched edge complicates the growth of a search tree, as discussed in Figure 11(a) below.) To discard B, we replace its contraction in the current graph $\overline{G}$ by the atoms and contracted blossoms of $C(B)$ and their incident edges. So after augmenting the matching and before proceeding to the next search, the algorithm does the following *discard step*:

Repeatedly discard a maximal blossom unless it is light and mature.

At the end of the discard step, every maximal blossom is light and mature (and still contracted). There can still be blossoms that are immature and/or heavy (like A' in Figure 7) but they are not maximal and so are "hidden," essentially irrelevant to the algorithm.

4.2 *b*-matching Algorithm

The b-matching algorithm differs from ordinary matching in at least three important ways: the search forest need not alternate at outer blossoms, a blossom step may create two blossoms (to hide a heavy blossom), and the expand step is more involved (to keep the search forest acyclic). We begin with terminology for the algorithm.

$\overline{G}$ denotes the graph with all blossoms contracted; $\overline{E}$ denotes its edge set. $\mathcal{S}$ denotes the search structure and $\overline{\mathcal{S}}$ is that structure in $\overline{G}$. Recall that in b-matching a vertex of V is not a blossom, so each vertex of $\overline{G}$ is an atom or a contracted blossom but not both. The notation B_x denotes the vertex of $\overline{G}$ containing $x \in V$; if x is atomic then $B_x = x$. For $x \in V$, if $d(x, M) < b(x)$ then B_x is *free*. (If B_x is a blossom we shall see that $d(x, M) < b(x)$ implies x is the base vertex of B_x.) The roots of the forest $\overline{\mathcal{S}}$ are the free atoms and free blossoms.

Let v be a node of $\overline{\mathcal{S}}$. v is *inner* if it is joined to its parent by an unmatched edge. Otherwise (i.e., v is joined to its parent by a matched edge, or v is a search tree root), v is *outer*. We refrain from classifying vertices contained in a blossom. (A vertex in an outer blossom can function as both inner and outer, because of its P_i-trails.)

As before, an edge is *tight* if it satisfies the LP complementary slackness conditions with equality (see Appendix B). Again, as before, we shall see that every matched edge is tight. The following notion identifies the edges that can be used to modify the search forest when they are tight. An edge $e = xy \in \overline{E} - \overline{\mathcal{S}}$ is *eligible for* B_x if any of the following conditions holds:

x is an outer atom and $e \notin M$;
B_x is an outer blossom;
B_x is an inner node and $e \in M$.

```
make every free atom or blossom an (outer) root of S̄
loop
    if ∃ tight edge e = xy ∈ Ē eligible for B_x with y ∉ S then
        /* grow step */
        add e, B_y to S
    else if ∃ tight edge e = xy ∈ Ē, eligible for both B_x and B_y then
        /* blossom step */
        if B_x and B_y are in different search trees then
            /* e plus the S̄-paths to B_x and B_y give an augmenting blossom B */
            augment M using B, and end the search
1       α ← the nca of B_x and B_y in S̄
        C ← the cycle S̄(α, B_x), e, S̄(B_y, α)
        if α is an inner node of S̄ then /* α is atomic */
2           contract C to a heavy blossom /* C is the new B_x */
            f ← the unmatched edge of S incident to α
            α ← the outer node of S̄ on f
            C ← the closed cycle f, B_x, f′
        if α is atomic and d(α, M) ≤ b(α) − 2 then /* α is a search tree root */
            augment M using blossom C, and end the search
3       contract C to an outer blossom /* C is the new B_x */
    else if ∃ inner blossom B with z(B) = 0 /* B is mature */ then
        /* expand step */
        define edges e ∈ δ(B, S̄ − M), f ∈ δ(B, M) /* f needn't be in S̄ */
        let P_0 be the trail e, P_0(v, β(B)), f, where v = e ∩ V(B), β(B) = f ∩ V(B)
        remove B from S /* this makes S̄ invalid if f ∉ S̄ */
        Expand (B, e, f) /* enlarge S̄ by a path through B to f */
    else adjust duals
```

Fig. 9. Pseudocode for a b-matching search.

The algorithm uses the following conventions:

> M denotes the current matching on G.
> For any edge e, e' denotes a new unmatched copy of e. e' always exists in b-matching.
> For related nodes x, y in $\overline{\mathcal{S}}$ (i.e., one of x, y descends from the other) $\overline{\mathcal{S}}(x, y)$ denotes the $\overline{\mathcal{S}}$-path from x to y.

The algorithm is presented in Figures 9 and 10. The next three subsections clarify how it works as follows. First we give some simple remarks. Then we state the invariants of the algorithm (which are proved in the analysis). Lastly, we give examples of the execution. These three subsections should be regarded as commentary—the formal proof of correctness and time bound is presented in Section 4.3.

Remarks. The grow step adds only one edge, unlike ordinary matching. One reason is that an inner vertex may have an arbitrary number of children: The possibility of >1 child comes from an inner atom on many matched edges. The possibility of no children arises when a new inner vertex has all its matched edges leading to vertices that are already outer.

Procedure Expand(B, e, f)
/* B: a blossom formed in a previous search, or an atom.
B is a mature blossom in the initial invocation
but may later be immature
e: edge entering B, already in $\overline{\mathcal{S}}$
f: edge leaving B
β: base vertex of B if B is a blossom
Expand enlarges $\overline{\mathcal{S}}$ by adding a path of edges of B that joins e to f
*/
if *B a blossom with $z(B) > 0$ or B an atom* /* e & f alternate */ **then**
4 make B a node of $\overline{\mathcal{S}}$
else if *B light and $e \in \delta(\beta, M)$* /* e.g., B light and P_0 contains $P_1(\beta, \beta)$ */ **then**
5 make B an outer node of $\overline{\mathcal{S}}$
else if $e, f \in \delta(\beta, E - M)$ **then**
/* B is heavy and P_0 contains trail $P_1(\beta, \beta)$ */
let $e = u\beta$
let C be the length two cycle B_u, e, B, e', B_u
if $d(u, M) \leq b(u) - 2$ /* u is a search tree root */ **then**
augment M along C and end the search
let B' be the blossom defined by $C(B') = C$
6 replace e in $\overline{\mathcal{S}}$ by outer node B'
else
7 let $P = (e_1, B_1, e_2, B_2, \ldots, e_k, B_k, e_{k+1})$ be the trail $E(P_0) \cap (C(B) \cup \{e, f\})$
/* $e_1 = e$, $e_{k+1} = f$, B_i a contracted blossom or an atom
P is a path
*/
for $i = 1$ **to** k **do**
if $i > 1$ **then** add e_i to $\overline{\mathcal{S}}$
Expand (B_i, e_i, e_{i+1})
for *every blossom B' of $C(B) - P$* **do**
repeatedly discard a maximal blossom in $V(B')$ unless it is light and mature

Fig. 10. Expand(B, e, f) for b-matching blossoms B.

A second reason comes from the definition of eligibility. It allows an outer vertex to have a child using a matched edge. So, an outer vertex may have an arbitrary number of children, using matched or unmatched edges. This also shows that the search forest need not alternate the same way as ordinary matching.

In the blossom step, the test $e \in \overline{E}$ is equivalent to the condition $B_x \neq B_y$ or $B_x = B_y$ is atomic. The second alternative allows a blossom whose circuit is a loop. This can occur for an atom that is either inner with a matched loop or outer with a tight loop. (Loop blossoms do not occur for blossoms B_x since our contraction operation discards loops.)

The contraction of line 3 creates an outer blossom that is light. When a heavy blossom is created in line 2 it gets absorbed in the light blossom of line 3. We shall see this is the only way a heavy blossom occurs in the search algorithm.

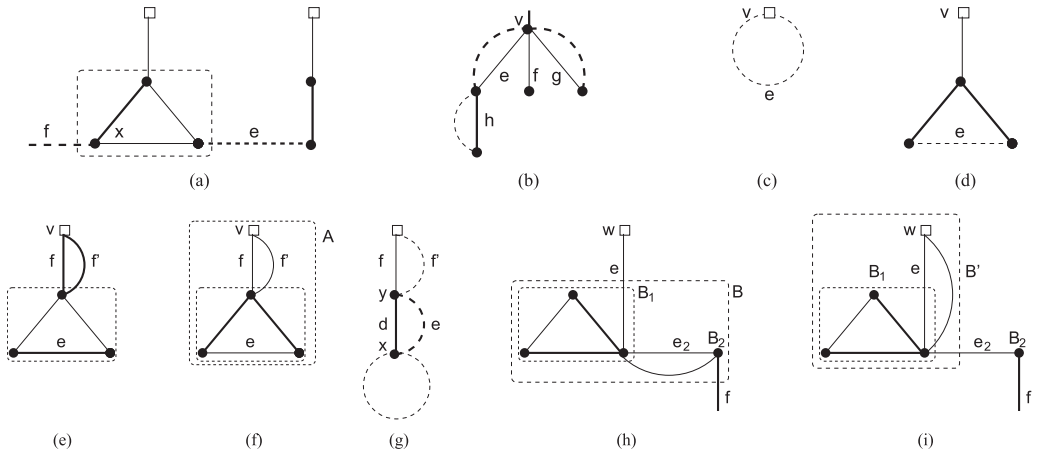

Fig. 11. Algorithm examples.

In the expand step, note that edge f may or may not be in $\overline{\mathcal{S}}$ (Figure 11(a) gives an example). This motivates the structure of Expand(B, e, f), which assumes on entry that e is an edge of $\overline{\mathcal{S}}$ but makes no assumption on f. Also, the trail P_0 is used for succinctness. An efficient implementation is described in the last subsection of Section 4.3.

If the duals are adjusted (last line of Figure 9), our assumption that the graph has a (perfect) b-matching guarantees the new duals allow further progress (i.e., a grow, blossom, or expand step can be done; this is proved in Lemma 4.13 and the discussion following it).

A tight edge xy with $B_x \in \overline{\mathcal{S}}$, is ignored by the algorithm in two cases: B_x an outer atom with $xy \in M$, and B_x inner with $xy \notin M$.

In procedure Expand, the last case (starting at line 7) corresponds roughly to the expand step for ordinary matching. The purpose of the preceding cases is to eliminate repeated vertices in P_0, which of course cannot be added to the search forest $\mathcal{S}$.

Invariants. The first several invariants describe the topology of $\overline{\mathcal{S}}$. As before, say that $\overline{\mathcal{S}}$ *alternates* at an $\overline{\mathcal{S}}$-node v if any edge to a child of v has opposite M-type from the edge to the parent of v; if v is a root then any edge to a child is unmatched. We can treat a root v as in the general case by using an artificial vertex ε as its parent (similar to augmenting trails): ε has an artificial matched edge to each atomic root as well as the base vertex of each blossom root.

(I1) $\overline{\mathcal{S}}$ alternates at any node that is not an outer blossom.

(I2) Let B be a maximal blossom with base vertex β. B is light.
If B is inner or not in $\mathcal{S}$ then it is mature.
If B is outer then any vertex $x \in V(B)$ has $d(x, M) = b(x)$ unless B is a root of $\overline{\mathcal{S}}$, $x = \beta$, and $d(x, M) = b(x) - 1$. If B is a nonroot of $\overline{\mathcal{S}}$ then β is on the matched edge leading to the parent of B.

(I3) For every blossom B (maximal or not), $C(B)$ is a cycle.

In detail, (I1) means the following: Any child of an inner $\overline{\mathcal{S}}$-node is outer. Any child of an outer atom is inner (this includes the case of an atomic search tree root). A child of an outer blossom may be outer or inner. Note that the first of these properties implies the parent of an inner node is outer. Equivalently, $\overline{\mathcal{S}}$ alternates at both ends of an unmatched edge in $\overline{\mathcal{S}}$.

In (I2), a "maximal blossom" is one that has been formed by the algorithm, at a given point in time not contained in any other. If B is inner then β is on a matched edge incident to B (since β is not free). If B is not in $\mathcal{S}$ then β may have such a matched edge or β may be free (immediately after an augment and before the next search begins).

The remaining invariants deal with the dual variables. (The duals for b-matching are reviewed in Appendix B.)

(I4) An edge is tight if it is matched, or it is an edge of $\overline{\mathcal{S}}$, or it is an edge of a contracted blossom.

(I5) A blossom with $z(B) > 0$ is light and mature.

Note that in (I5), blossom B need not be maximal.

Examples. Grow Step: In Figure 11(a), the triangular blossom B_x is inner. Its incident matched edge e leads to a node that is already outer. So unlike ordinary matching B_x has no corresponding grow step. B_x also illustrates the possibility of an expand step where the inner blossom is a leaf.

(I2) requires the inner blossom B_x to be mature. Suppose it was immature. A matched edge like f incident to B_x might exist. A grow step for f would be invalid, since $\mathcal{S}$ does not contain an alternating trail from f to a free vertex. Requiring inner blossoms to be mature avoids this complication.

Blossom Step: In Figure 11(b), v is an outer vertex. Suppose v is a contracted blossom. A blossom step can be done for the matched copy of e, as well as the matched copy of g. The unmatched copy of h is necessarily tight. A blossom step can be done for it, after e. This process illustrates how outer blossoms can be enlarged to become mature. (Edge e of Figure 11(g) illustrates another case.) On the other hand, if v is an atom none of these blossom steps can be done.

In Figure 11(c), the unmatched loop e may not be tight. If it becomes tight it forms an augmenting blossom if the free vertex v lacks at least two matched edges. The algorithm can repeatedly match copies of e as long as this condition holds. If eventually v lacks one matched edge, another copy of e forms a light blossom.

Figure 11(d) shows a search tree when a blossom step for e is discovered. The triangle is made a heavy blossom. If v lacks at least two edges the matching is augmented (Figure 11(e)). The discard step then abandons the triangle blossom. (Newly matched edges f, f' show the blossom is immature.) If v lacks only one edge the light blossom, A of Figure 11(f) is formed. This illustrates how the algorithm never creates heavy blossoms that are maximal.

In Figure 11(g), a blossom step creates the loop blossom B_x. Edge e is a matched copy of the search tree edge d. A blossom step may be done for e, B_x and y. (y is necessarily an atom because of unmatched edges d, e.) The unmatched edges f, f' give an augmenting trail or blossom as in Figure 11(e) and (f).

Expand Step: In Figure 11(h), blossom B is inner and mature. As an example of how this occurs, blossom B and atom B_2 may be, respectively, blossom A of Figure 11(f) and its base vertex v. In Figure 11(f), blossom A gets positive z when duals are adjusted. An augment is done using an unmatched edge of $\delta(v) \cap \delta(A)$, so A does not change. v is no longer free. A then becomes inner in a grow step from the free vertex w of Figure 11(h).

A dual adjustment makes $z(B) = 0$ and B is expanded. Line 7 generates two recursive calls, with arguments B_1, e, e_2 and B_2, e_2, f. (Note $z(B_1)$ must be 0.) The first recursive call either augments the matching as in Figure 11(e) or forms a blossom as in Figure 11(f). In the second case, Figure 11(i) gives the result of the entire expand step.

Finally, suppose in Figure 11(h) blossom B is not maximal: It is included in a blossom B_3 (not shown), where $\alpha(B_3) = B$, $C(B_3)$ is a length two closed trail containing B and an atom x, with two

copies of edge vx, say h and i where $h \in M$, $i \notin M$. The initial call to Expand issues one recursive call with arguments B, e, f. The rest of the expand step is as before. It results in Figure 11(i) with x no longer in a blossom. x can then be added to blossom B' (via a grow and a blossom step, for edges h and i, in either order).

4.3 Analysis

The analysis is presented in three subsections. First we prove the invariants. Then we prove the algorithm is correct, adding some details about initialization and termination. Lastly, we prove the desired time bound, adding some implementation details for dual variables.

Proof of Invariants

We show that all the invariants are preserved by every step of the algorithm. The first four lemmas treat grow steps, blossom steps, augment steps, and expand steps, respectively. (The lemmas for the blossom step and the expand step assume any augment is done correctly; the augment step lemma treats the detailed augment and discard steps of Section 4.1.) Then we check dual adjustment. Many details are straightforward, so we only discuss the most interesting and representative cases.

Lemma 4.7. *The grow step preserves every invariant.*

Proof. We will verify the case $e \in M$ and B_y a blossom.

The algorithm shows e is eligible, so B_x is either an outer blossom or an inner node. Thus (I1) holds. B_y starts out not in $\mathcal{S}$, so (I2) shows it is mature. Vertex y must be the base of B_y. Thus (I2) holds for the new outer blossom B_y. (I4) shows e is tight, so adding it to $\overline{\mathcal{S}}$ preserves (I4). (I3) and (I5) are unchanged. □

Lemma 4.8. *The blossom step preserves every invariant.*

Proof. We will verify the case of a blossom step that starts with α in line 1 an outer node, and contracts C in line 3. (The other cases are similar to this one.)

Since α is outer, the contracted C is outer. Thus (I1) is vacuous for C. (I4) holds since the edges of C move from $\overline{\mathcal{S}}$ to the contracted blossom. (I5) does not change. We check (I2), (I3), and the fact that C is a valid blossom, as follows:

(I3): C is the fundamental cycle of e in the forest $\overline{\mathcal{S}}$.
(I2): We check (I2) for the outer blossom C.

Case *α is an atom*: α is the base vertex β of the blossom C. To show C is light, observe that at least one of the two edges of $\delta(\alpha, C)$ goes to a child of α. (I1) shows the edges at the outer atom α alternate, so that edge is unmatched. Thus C is light.

The test preceding line 3 ensures $d(x, M) \geq b(x) - 1$ for $x = \alpha = \beta$; $d(x, M) = b(x)$ for the remaining vertices of $V(C)$ since no atom or blossom of $C - \alpha$ is free. If α is a root of $\overline{\mathcal{S}}$ clearly it has $d(x, M) = b(x) - 1$. If α is not a root it has the desired matched edge to the parent of C.

Case *α is a blossom*: (I2) shows α is light, so C is light. The other properties follow the atomic case.

C satisfies Definition 4.2 of a blossom: Note that α and C of the definition are the same as α and C of the algorithm. The conditions to verify are for the three types of vertices v of cycle C:

Case *v is an atom and $v = \alpha$*: An edge $e \in \delta(\alpha, C)$ either goes to a child of α or e is the blossom edge joining B_x and B_y. In both cases, e is eligible for the outer atom α, so $e \notin M$. Thus, the first and last edges of C have the same M-type.

Case v *is an atom,* $v \neq \alpha$: If $v \notin \{B_x, B_y\}$, then (I1) shows the two edges of $\delta(v, C)$ alternate. If $v \in \{B_x, B_y\}$, then one edge of $\delta(v, C)$ goes to its parent and the other is eligible for v. The two cases of eligiblity for an atom show the edges of $\delta(v, C)$ alternate.

Case v *is a contracted blossom*: C is a cycle so $d(v, C) = 2$. The rest of the verification concerns the case $v \neq \alpha$. Let β be the base vertex of v. (I2) shows v is light, so the definition requires β to be on a matched edge of C.

If v is outer, (I2) shows the matched edge f going to v's parent is incident to β. Furthermore, $f \in C$.

If v is inner, (I2) shows it is mature. Thus v is on a unique matched edge f, which is incident to β. If $v \notin \{B_x, B_y\}$, then C passes through a child of v. f goes to the unique child of v, so $f \in C$. If $v \in \{B_x, B_y\}$, then edge e of the blossom step is eligible for v, so by definition $e \in M$. Thus, $e = f$ and $f \in C$. □

In the next two lemmas, some invariants are explicitly violated during the execution of a step. However, we only require the invariants to be satisfied at the end of the step, assuming they hold at the start.

Lemma 4.9. *The augment and discard steps preserve every invariant.*

Proof. Consider the rematching of the augment step. It preserves (I5) by Lemma 4.6. (I4) holds since the rematching is done along a P_i trail of the augmenting blossom. (I3) is unchanged. (I1) is vacuous after the augment step, the search forest $\overline{\mathcal{S}}$ is empty.

The rematching can violate (I2), since a maximal blossom that is incident to two matched edges on the augmenting trail becomes heavy. Also, there may be maximal blossoms that are light and immature (and not in $\mathcal{S}$, which is empty). This violation is only temporary since the discard step eliminates any maximal blossom that is not light and mature. Thus, (I2) holds at the start of the next search. The discard step also preserves (I4), since (I5) shows no blossom with positive z is eliminated, and thus the set of tight edges does not change. □

We turn to the expand step. Recall the main issue is preserving $\overline{\mathcal{S}}$ as a forest (i.e., no repeated vertices). The strategy of Figure 10 is based on invariant (I3). (I3) implies that in any trail $P_i(v, \beta(B))$, the repeated vertices are precisely the blossom bases β that have $P_1(\beta, \beta)$ as a subtrail. (This is proved by a simple induction.) Our algorithm contracts these subtrails into outer blossoms (lines 5 and 6).

To begin the formal analysis, let $\mathsf{Expand}(B, \dot{e}, \dot{f})$ denote the call of the expand step in Figure 9. Recall that $\overline{\mathcal{S}}$ is defined as a forest rooted at the free atoms and blossoms. When the expand step removes B from $\mathcal{S}$, this definition and (I1) will be violated if $\overline{\mathcal{S}}$ has a nonempty subtree descending from $\dot{f}$. (Unlike ordinary matching this subtree needn't exist, e.g., Figure 11(a).)

To remedy this, we modify (I1) to a condition (I1′) defined as follows. At any point in the execution of Expand let $\overline{\mathcal{P}}$ be the set of edges of the trail P_0 (defined in the expand step) that have been added to $\overline{\mathcal{S}}$ so far. Say that a trail in $\overline{\mathcal{S}}$ has *permissible alternation* if it alternates at any node that is not an outer blossom. Consider an arbitrary (perhaps recursive) invocation $\mathsf{Expand}(B, e, f)$.

(I1′) If on entry to $\mathsf{Expand}(B, e, f)$, $\overline{\mathcal{P}}$ is a trail that has permissible alternation and has e as its last edge, then on exit $\overline{\mathcal{P}} + f$ (for the new $\overline{\mathcal{P}}$) has permissible alternation and has f as its last edge.

Note that, in general, the exit trail $\overline{\mathcal{P}}$ is an extension of the entry trail. The exit condition does not mean that f belongs to $\overline{\mathcal{P}}$.

Say $\mathsf{Expand}(B, e, f)$ is a *base case execution* if it does not recurse, i.e., it executes line 4, 5, or 6.

Claim. *If (I1′) holds for every base case execution of* Expand, *then the expand step ends with* $\overline{\mathcal{P}}$ *having permissible alternation and joining* $\dot{e}$ *to the end of* $\dot{f}$ *in* $V(B)$.

Remark. If $\dot{f}$ was an edge of $\overline{\mathcal{S}}$ at the start of the expand step, the claim implies $\dot{f}$ and its descendants are once again connected to a root of $\overline{\mathcal{S}}$.

Proof. Let Expand(B, e, f) be a base case execution with Expand(B, e', f') the next base case execution. Observe that when Expand(B, e', f') is entered $e' = f$ is the last edge added to $\overline{\mathcal{S}}$. This follows by a simple induction using the structure of the recursive calls when line 7 is executed. The observation implies that if the call to Expand(B, e, f) satisfies the exit condition of (I1′) then the entry condition of (I1′) holds for Expand(B, e', f').

The initial call Expand$(B, \dot{e}, \dot{f})$ starts with $\overline{\mathcal{P}} = \dot{e}$. So, stringing together all the base case executions shows that when Expand$(B, \dot{e}, \dot{f})$ exits, $\overline{\mathcal{P}} + \dot{f}$ has permissible alternation and has $\dot{f}$ as its last edge. This gives the claim. □

Lemma 4.10. *The expand step preserves every invariant.*

Proof. We examine the four possibilities for an execution of Expand, lines 4–7. We will show the three base cases satisfy (I1′) and also preserve invariants (I2)–(I5). In the recursive case line 7 we will verify that $\overline{\mathcal{P}}$ is a path, i.e., no repeated vertices. This will complete the verification of (I1).

Case *line 4 is executed*: First, we show that e and f alternate (as in the comment). The test guarding line 4 ensures B is atomic or mature, by (I5). Line 4 is not executed in the initial call Expand$(B, \dot{e}, \dot{f})$. So the current invocation was made from line 7, as Expand(B_i, e_i, e_{i+1}) for some $i \le k$. The rest of the argument switches to this parent invocation, where we must show that e_i and e_{i+1} alternate when B_i is atomic or mature.

When $i \notin \{1, k\}$, e_i and e_{i+1} are the edges of $\delta(B_i, C(B))$. The edges alternate for atomic B_i by the definition of blossom. For mature B_i we use that definition and the definition of maturity. When i is 1 or k (i.e., $e_i = e$ or $e_{i+1} = f$, or both) a similar inspection, using the definitions of P and P_0, shows e_i and e_{i+1} alternate.

The alternation of e and f implies (I1′). For (I2), a blossom B is light by (I5). The properties of (I2) for inner and outer B follow easily. (I3)–(I5) are unchanged.

Case *line 5 is executed*: $e \in M$ makes B outer. So (I1′) holds trivially for f and B. The other invariants hold trivially. (Note that B may be immature.)

Case *line 6 is executed*: It is easy to see B' is a valid blossom. To establish (I2), first observe that B' is light: If u is atomic, then $e \notin M$ makes B' light with base u. If B_u is a contracted blossom, it is light by (I2) and again B' is light.

Next, observe B' is outer: e is unmatched and in $\overline{\mathcal{P}}$ and $\overline{\mathcal{S}}$ when (this invocation of) Expand starts. So, B_u is outer by the permissible alternation of e. This makes B' outer. The other properties of (I2) as well as (I3)–(I5) follow easily. (I1′) holds as in the previous case.

Case *line 7 is executed*: We claim line 7 is never executed for P a nonsimple trail, i.e., $P = P_1(\beta(B), \beta(B))$. In proof, consider an invocation Expand(B, e, f), where $C(B)$ is traversed this way. If B is light then e and f are matched and both incident to $\beta(B)$. So, line 5 is executed. If B is heavy, then e and f are unmatched and both incident to $\beta(B)$. So line 6 is executed.

Recall that (I3) implies the only P_i-trail that repeats a node of $C(B)$ is $P_1(\beta, \beta)$. Thus, P is a path in line 7, and the recursive calls for line 7 keep $\overline{\mathcal{P}}$ and $\overline{\mathcal{S}}$ acyclic.

The discard step of this case ensures (I2). The remaining invariants hold by induction. □

We turn to the dual adjustment step. The following lemma and its corollary describe $\overline{\mathcal{S}}$ when duals are adjusted.

Lemma 4.11. *When duals are adjusted, $\overline{\mathcal{S}}$ alternates at every node and every maximal blossom is light and mature.*

Proof. $\overline{\mathcal{S}}$ alternates at every node when every blossom is mature. This follows from (I1) plus the observation that (I2) prohibits a mature outer blossom being joined to a child by a matched edge. (I2) also shows that every maximal blossom is light. Thus to prove the lemma we need only show that every maximal blossom is mature when duals are adjusted. (I2) shows this for inner and non-$\mathcal{S}$ blossoms. So we need only consider outer blossoms.

Let B be an immature outer blossom. We will show that a grow or blossom step can be done for B. This proves the lemma, since duals are adjusted only when no such step can be done.

Let β be the base vertex of B. Let $f \in \delta(\beta, M)$ be the edge leading to the parent of B in $\overline{\mathcal{S}}$; f is undefined if B is free. (I2) implies either B is free and every $x \in V(B)$ has $d(x, M) = b_\beta(x)$ or f exists and every $x \in V(B)$ has $d(x, M) = b(x)$. In both cases, B immature implies there is an edge $e = xy \in \delta(B, M - f)$ with $x \in V(B)$. Clearly e is tight.

Case $B_y \notin \overline{\mathcal{S}}$: Since B_x is an outer blossom, e is eligible for B_x. Thus, a grow step can be done to add B_y to $\mathcal{S}$.

Case $B_y \in \overline{\mathcal{S}}$: Suppose B_y is inner. Then $e \notin \overline{\mathcal{S}}$. ($e \in \overline{\mathcal{S}} - f$ makes B_y a child of B_x, so B_y is outer.) As before, e is eligible for B_x, and $e \in M$ shows it is eligible for B_y. So, a blossom step can be done for e.

Suppose B_y is outer. Any matched edge is tight, so the unmatched copy e' of e is tight. $e' \notin M$ is eligible for both outer vertices B_x, B_y. So, a blossom step can be done for e'. □

In contrast to ordinary matching, duals may be adjusted with matched edges xy not in $\overline{\mathcal{S}}$ but incident to nodes of $\overline{\mathcal{S}}$: x and y can be atoms with x inner and y outer. This possibility is governed by the following lemma.

Corollary 4.12. *When duals are adjusted, any matched edge incident to a node of $\overline{\mathcal{S}}$ joins an inner node to an outer node.*

Proof. Take $e = xy \in M$ with B_x a node of $\overline{\mathcal{S}}$.

Case B_x *is inner*: Since no grow step can be done, $y \in \mathcal{S}$. If $e \in \overline{\mathcal{S}}$, it goes to a child of B_x, which is outer as claimed. If $e \notin \overline{\mathcal{S}}$, since no blossom step can be done B_y is an outer atom, as claimed.

Case B_x *is outer*: An unmatched copy e' of e is tight and not in $\overline{\mathcal{S}}$. Since no grow step can be done for e', $y \in \mathcal{S}$. Since no blossom step can be done for e', B_y is inner, as claimed. □

Let us check that the invariants are preserved when duals are adjusted. (I1)–(I3) are unaffected by the adjustment. For (I4)–(I5), recall the dual adjustment step (Figure 19 of Appendix B).

(I4): Edges of $\overline{\mathcal{S}}$ remain tight when duals are adjusted, by the alternation of the lemma. Matched edges incident to $\overline{\mathcal{S}}$ remain tight, by the corollary. Edges in contracted blossoms remain tight by definition of the dual adjustment. Thus (I4) is preserved.

(I5): A dual adjustment only increases the duals of maximal blossoms (that are outer). So the lemma implies (I5) for the blossoms whose z value increases from 0 to positive.

We have shown the dual adjustment preserves all the invariants. Furthermore, the dual variables continue to be valid, i.e., they are feasible for the b-matching linear program (reviewed in Appendix B). We have verified this for all cases except unmatched edges not in a blossom subgraph. This case follows by exactly the same argument as ordinary matching.

Termination, Correctness, and Initialization

As in ordinary matching, each step of the algorithm makes progress—it either finds an augmenting trail or modifies the graph in a way that the current search will not undo. (Each nonaugmenting step creates a new node of $\overline{\mathcal{S}}$. Obviously, the number of new outer atoms, outer blossoms, or inner atoms is limited. The number of new inner blossoms is also limited—a given blossom from a previous search can become an inner blossom only once.) Thus, the algorithm does not loop, it eventually halts.

Next we show the algorithm halts with a perfect matching. We prove this using the fact that the maximum size of a b-matching is

$$\min_{I\subseteq V}\left(b(I)+\sum_{C}\lfloor b(C)/2\rfloor\right), \tag{4.3}$$

where C ranges over all nontrivial connected components of $G-I$ (a component is trivial if it consists of one vertex x but no loop, i.e., $xx\notin E$) [34, Theorem 31.1]. In fact, it is straightforward to see that the above quantity upper-bounds the size of a b-matching (a matched edge either contains a vertex of I or has both vertices in a component C). Our derivation gives an alternate proof that the bound is tight.

Consider a search that *fails*, i.e., no grow, blossom, or expand step is possible, and duals cannot be adjusted to remedy this.

Lemma 4.13. *In a failed search, any edge $e\in E$ incident to an $\mathcal{S}$-vertex is either spanned by a blossom or incident to an inner atom.*

Proof. A dual adjustment decreases the z-value of any inner blossom, and lowering it to 0 allows an expand step to be done. So there are no inner blossoms.

We can assume $e\notin M\cup\overline{\mathcal{S}}$ by using an unmatched copy. Let $e=uv$. If the lemma does not hold, we can assume at least one of B_u, B_v is outer, say B_u. Furthermore, either $B_v\notin\overline{\mathcal{S}}$, or B_v is outer with either $B_u\neq B_v$ or $B_u=B_v$ atomic (since e is not spanned by a blossom). Since no grow or blossom step can be done, e is not tight, in each of these cases. A dual adjustment decreases the y-value of any vertex in an outer node. So, duals can be adjusted to make e tight. This makes a grow or blossom step possible, contradiction. □

Consider a failed search. Let I be the set of inner atoms. The lemma shows that deleting I gives a collection of connected components, each of which is either (a) an outer blossom, (b) an outer atom that has no loop, or (c) a set of non-$\mathcal{S}$ vertices. (For (b), note that the existence of a loop ensures the vertex is in a blossom, by the lemma.) Corollary 4.12 shows no matched edge joins two vertices of I. Observe that

$$|M|=b(I)+\sum_{C}\lfloor b(C)/2\rfloor,$$

where the sum ranges over the components of type (a) or (c). (For (a), Lemma 4.11 shows the blossom is mature. For (c), note that the non-$\mathcal{S}$-vertices are perfectly matched, i.e., each $v\notin\mathcal{S}$ is on $b(v)$ matched edges leading to other non-$\mathcal{S}$-vertices.) This shows the upper bound (4.3) is tight. It also shows that our algorithm, executed on an arbitrary input graph, halts with a maximum cardinality b-matching.

When a perfect b-matching exists our algorithm finds such a matching of maximum weight. This follows simply from the LP formulation of maximum b-matching of Appendix B. The argument is the same as ordinary matching, so we just summarize it as follows.

The primal LP is satisfied by any (perfect) b-matching. The dual LP requires every edge to be dual-feasible, i.e.,

$$\widehat{yz}(e) = y(e) + z\{B : e \subseteq B\} \geq w(e).$$

The dual adjustment step enforces this. Complementary slackness requires tightness in the above inequality for every $e \in M$. Complementary slackness also requires every blossom with positive z to be mature. These two complementary slackness conditions are guaranteed by (I4) and (I5), respectively. We conclude the final b-matching has maximum weight, and our algorithm is correct. In fact, the algorithm provides an alternate proof that the LP of Appendix B is a correct formulation of maximum b-matching.

As in ordinary matching, the algorithm can be advantageously initialized to use any information at hand. The initialization must specify a partial b-matching M, dual functions y, z, and a collection of blossoms $\mathcal{B}$. The requirements are that the duals must be feasible on every edge, tight on every edge of M or a blossom subgraph, and invariants (I2), (I3), and (I5) must hold. The simplest choice is a function y on vertices where $y(e) \geq w(e)$ for every edge e, $z \equiv 0$, $\mathcal{B} = \emptyset$, and M a partial b-matching consisting of tight edges. (Here and elsewhere $z \equiv 0$ means z is the function that is 0 everywhere.) This initialization is used in Section 4.4. A handy special case is when every vertex is assigned the same initial y-value. This gives the invariant that every free vertex has the same y-value, which is the minimum y-value. Using this initialization when the input graph does not have a perfect b-matching, our algorithm finds a *maximum cardinality maximum weight b-matching*, i.e., a partial b-matching that has the greatest number of edges possible, and subject to that constraint, has the greatest weight possible. This is shown in Appendix B, along with other maximum weight variants. Other choices for initialization allow various forms of sensitivity analysis to be accomplished in $O(1)$ searches after finding a maximum b-matching (as in ordinary matching).

Dual Variables and Efficiency Analysis

The numerical computations of the algorithm are organized around a parameter Δ maintained as the total of all dual adjustment quantities δ in the current search. (δ is computed by the dual adjustment algorithm of Figure 19, Appendix B.) We use Δ as an offset to compute dual variables as they change, and in a data structure to adjust duals and determine the next step to execute. The details are as follows.

The data structure records values $Y(v), Z(B)$ that are used to find the current value of any dual $y(v), z(B)$ ($v \in V, B$ a blossom) in $O(1)$ time. For example, let v be a vertex currently in an outer node of $\overline{\mathcal{S}}$, with Δ_0 the smallest value of Δ for which this is true, and $y_0(v)$ the value of $y(v)$ at that point. (Δ_0 may be less than the value of Δ when the current blossom B_v was formed.) Then

$$\begin{aligned} Y(v) &= y_0(v) + \Delta_0, \\ y(v) &= Y(v) - \Delta, \end{aligned}$$

since $y(v)$ decreases by δ in every dual adjustment where it is outer. Similarly, a blossom B that is currently an outer node of $\overline{\mathcal{S}}$ has

$$\begin{aligned} Z(B) &= z_0(B) - 2\Delta_0, \\ z(B) &= Z(B) + 2\Delta, \end{aligned}$$

for Δ_0 the value of Δ when B became outer and $z_0(B)$ the value of $z(B)$ at that point. ($z_0(B)$ is the value of $z(B)$ at the start of the search if B was formed prior to that, or 0 if B was formed in the current search.) Modulo changes of sign, similar equations are used to compute $y(v)$ for v

currently an inner atom and $z(B)$ for B currently a maximal inner blossom. Other y and z values are computed as described in Appendix D.

To adjust duals and determine the next step of the algorithm to be executed, we use a Fibonacci heap $\mathcal{F}$. $\mathcal{F}$ contains a node for each grow, blossom, and expand step that is a candidate for the next step to execute. The key of each such node is the (future) value of Δ when the step can be done. Specifically, when a blossom B becomes inner in a grow or expand step, a node for expanding B is inserted in $\mathcal{F}$ with key equal to the current value of Δ plus $z(B)/2$. (This node may get deleted before the expansion, in a blossom step.) Theorem 3.5 provides the node of $\mathcal{F}$ for the next candidate blossom step. Gabow [14] gives an algorithm for future grow steps that uses total time $O(m\alpha(m, n))$. For completeness, Appendix D gives a simpler algorithm for grow steps. It uses total time $O(m + n \log n)$ and so suffices for our purposes. The algorithm is also valid for a pointer machine.

The algorithms of Figures 9 and 10 use linear time. We use the data structure for blossoms of Appendix C. Note that in the expand step of Figure 9 P_0 should not be computed:[9] Instead, the various paths P in line 7 of Figure 10 are essentially precomputed, as shown in Appendix C.

Blossom steps are implemented using the tree-blossom-merging algorithm of Section 3.1. This algorithm is unchanged from ordinary matching, assuming the supporting forest is maintained correctly. (A new case is that matched edges can cause blossom steps, and such edges may be incident to inner vertices. Such blossom steps are handled seamlessly by the tree-blossom-merging algorithm. Alternatively they can be easily handled outside that algorithm, since matched edges are always tight.)

Now consider the supporting forest. We modify Definition 3.1 of the supporting forest, which uses the paths $P(x, \beta)$ of ordinary matching for inner blossoms B. The corresponding trail for b-matching is $P_0(x, \beta)$. To keep the supporting forest acyclic we modify this trail as follows. Let $P_0^-(x, \beta)$ be $P_0(x, \beta)$ with every maximal subtrail of the form $P_1(\beta(A), \beta(A))$ replaced by the vertex $\beta(A)$. We modify Definition 3.1 so that inner blossoms B use $P_0^-(x, \beta)$ as T_B rather than $P(x, \beta)$.

Note that an inner blossom B still has $\beta(B)$ a vertex of the supporting forest. This allows a blossom step for the matched edge incident to $\beta(B)$ to be handled correctly.

The algorithm for maintaining T is essentially unchanged for grow and blossom steps. For expand steps suppose Expand is executed for a blossom A, which as above is represented only by the vertex $\beta(A)$. We cannot have $z(A) > 0$. (That would make A light and mature, by (I5). But maturity implies a subtrail $P_i(\beta(A), \beta(A))$ of $P_0(x, \beta)$ has $i = 0$, contradicting $i = 1$.) So, line 5 or 6 is executed, making A a new outer vertex. $\beta(A)$ is already in the supporting tree, and the remaining vertices of A are added to T using *add_leaf* operations.

We conclude that our algorithm is correct and achieves the desired efficiency:

THEOREM 4.14. *A maximum b-matching can be found in time $O(b(V)(m + n \log n))$.*

4.4 Strongly Polynomial Algorithm

To extend the algorithm to a strongly polynomial version, we follow previous approaches. They use bipartite matching (i.e., network flow) to reduce the size of the given nonbipartite problem [3, 23, 34]. The high-level algorithm is as follows:

[9] The pseudocode uses P_0 to facilitate specification of P. Explicitly computing P_0 could lead to quadratic time when there is a nesting of blossoms that get expanded, i.e., we have $B = B_0 \supset B_1 \supset B_2 \supset \ldots$ where each B_i becomes inner when B_{i-1} gets expanded.

Set $b' = 2\lfloor b/2 \rfloor$. Let M be a maximum cardinality maximum weight b'-matching with corresponding optimal dual function y. Using M, y (and $z \equiv 0,\ \mathcal{B} = \emptyset$) as the initial solution, execute the b-matching algorithm of Section 4.2.

This is a straightforward combination of previous algorithms [3, 23, 34]. For completeness, we give the analysis. Correctness of the algorithm—that an optimum M, y actually exists—may not be immediately clear. We establish correctness below as part of the efficiency analysis.

Since we assume G has a perfect b-matching, it has a partial b'-matching with $\geq \frac{b(V)}{2} - n$ edges. So, our b-matching algorithm performs $\leq n$ augmentations. Thus, the time for the entire algorithm is $O(n(m + n \log n))$ plus the time to find M, y. We will see the latter strictly dominates the time.

We find M, y as follows. Extend the given graph G to G^+ by adding a vertex s with $b'(s) = b'(V)$, edges vs, $v \in V$ of weight 0 and edge ss of weight $Wb'(s)$ for

$$W = \max\{1, |w(e)| : e \in E(G)\}.$$

It is easy to see there is a 1-1 correspondence between partial b'-matchings of G and (perfect) b'-matchings of G^+, wherein a cardinality c partial matching corresponds to a perfect matching with c loops ss. Furthermore, a maximum cardinality maximum weight b'-matching of G corresponds to a maximum b'-matching of G^+. To verify this last assertion, it suffices to show any b'-matching of G^+ with c loops ss, say M_c, weighs more than any such matching with $d < c$ loops, say M_d. This follows since the relation $b'(s)/2 \geq c \geq d + 1$ gives

$$\begin{aligned} w(M_c) &\geq (Wb'(s) - W)c \geq (Wb'(s)d + Wb'(s)) - Wb'(s)/2 \\ &= Wb'(s)d + Wb'(s)/2 > (Wb'(s) + W)d \geq w(M_d). \end{aligned}$$

We find a maximum b'-matching of G^+ by reducing to a bipartite graph BG. BG has vertex set $\{v_1, v_2 : v \in V(G^+)\}$, edge set $\{u_1v_2, u_2v_1 : uv \in E(G^+)\}$, and edge weights and degree constraints given, respectively, by

$$w(u_1v_2) = w(u_2v_1) = w(uv) \text{ and } b'(v_1) = b'(v_2) = b'(v)/2.$$

(Note s has even b' value. Also, a loop uu of G^+ gives one edge u_1u_2 in BG.) Let x be a maximum b'-matching on BG with optimum dual function y (we show x and y exist below). Define a b'-matching M on G^+ by taking $x\{u_1v_2, u_2v_1\}$ copies of each edge $uv \in E(G^+)$ (by our summing convention, this means a loop uu has $x(u_1u_2)$ copies). Define a dual function y by

$$y(v) = y\{v_1, v_2\}/2. \tag{4.4}$$

We will show that restricting M and y to G gives the optimum values desired for the main algorithm.

We first prove that the b'-matching x exists, and M is a maximum b'-matching on G^+. These properties will follow from

(a) any b'-matching x on BG gives a b'-matching M on G^+ of the same weight;

(b) any b'-matching M on G^+ gives a b'-matching x on BG of the same weight.

Recall that G^+ has a b'-matching. So (b) implies x exists. Also, (a) is obvious from the above construction of M on G^+. So, we need only prove (b).

We prove (b) using the Euler tour technique: Let M be a b'-matching on G^+. Since b' is even on G^+, the edges of M form a collection of closed trails. Traverse each trail, and for each edge uv traversed from u to v match edge u_1v_2. This applies to loops uu too. For each vertex v, the BG-matching has exactly $b'(v)/2$ edges incident to each of v_1, v_2. Clearly we have the desired matching on BG.

Applying complementary slackness to the definition of the dual function for bipartite b-matching [34, chap. 21] we get that a b'-matching x on BG and a dual function y are both optimum

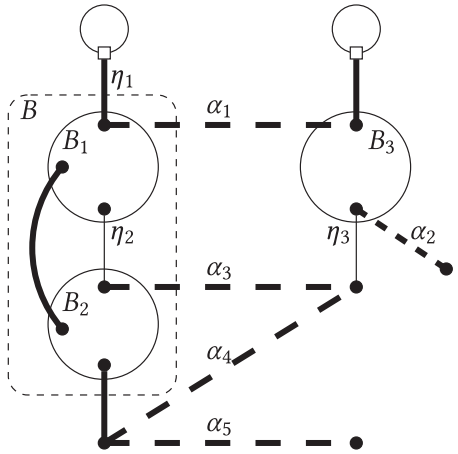

Fig. 12. f-factor search structure. The dashed edges α_i are not part of the structure. They are all matched.

iff

$$y(e) \geq w(e) \text{ for all edges } e \text{ of } BG, \text{ with equality when } x(e) > 0. \tag{4.5}$$

(Recall our summing convention means that if $e = uv$, then $y(e) = y(u) + y(v)$.) Thus, for every edge $e = uv$ of G^+,

$$y(e) = (y(u_1) + y(u_2) + y(v_1) + y(v_2))/2 \geq (w(u_1v_2) + w(u_2v_1))/2 = w(e).$$

Furthermore, equality holds for $e \in M$. This follows because the matching x on BG has a mirror image x' defined by $x'(a_1b_2) = x(b_1a_2)$. Thus $x(u_1v_2) > 0$ implies $y(u_1v_2) = w(e)$ as well as $y(v_1u_2) = w(e)$. This calculation remains valid when e is a loop uu (i.e., $u = v$). So the functions $y, 0$ are optimum duals for b'-matching on G^+. Restricting M and y to G, and taking $z \equiv 0$, $\mathcal{B} = \emptyset$, gives permissible initial values for the b-matching algorithm of Section 4.2. (Recall the discussion of initialization at the end of Section 4.2.) We have proved the main algorithm is correct.

We find x, y on BG using an algorithm for minimum cost network flow. Specifically, the problem on BG is a transportation problem, where x is an optimum integral solution and y is an optimum dual function [34, chap. 21]. The optimality conditions, Equation (4.5), are precisely those for the transportation problem (assuming the trivial sign flip to convert our maximum weight problem to a minimum cost problem). Orlin solves the transportation problem (more generally, the transhipment problem) in time $O(n \log n(m + (n \log n))$ [29]. It gives both x and y. Using this, we obtain our strongly polynomial bound:

Theorem 4.15. *A maximum b-matching can be found in time $O(\min\{b(V), n \log n\}(m + n \log n))$.*

5 f-FACTORS

The fundamental difference between f-factors and b-matching is illustrated in Figure 12, which shows a search structure for f-factors. Recall that f-factors are defined on arbitrary multigraphs—unlike b-matching, edges have a limited number of parallel copies. The parallel copies in b-matching often allow a blossom to be enlarged using an incident matched edge and its parallel unmatched edge (e.g., in Figure 12 blossom B and α_3; also B_3 and α_2). The parallel unmatched edge needn't exist for f-factors (in fact, the search structure of Figure 12 might be maximal). This in turn leads to another major difference: the linear programming z dual variables are assigned to blossom/incident-edge-set pairs rather than just blossoms.

The organization of the f-factor section is the same as b-matching: Section 5.1 gives the basic properties of blossoms. Section 5.2 presents our algorithm. Section 5.3 gives the analysis, proving we find a maximum f-factor in time $O(f(V)(m + n \log n))$. Section 5.4 extends the algorithm to achieve the strongly polynomial time bound $O((m \log n)\ (m + n \log n))$, the same bound as known for bipartite graphs.

We use the same terminology as b-matchings whenever possible—we introduce each such duplicate term and point back to its complete definition in Section 4. For instance, the degree-constraint function f, a *partial f-factor*, and all terminology regarding multigraphs and contractions are the same as before.

5.1 Blossoms

Similar to b-matching, we define immature and mature blossoms, and show how blossoms are updated when the matching gets augmented.

Blossoms are defined as before by Definition 4.2. We add the notion of "base edge" of a blossom. It is the "exiting edge." (It is used implicitly for b-matchings.)

Definition 5.1. The *base edge* of a blossom A, denoted $\eta(A)$, is either an edge of $\delta(\beta(A)) \cap \delta(V(A))$ with opposite M-type from A, or $\emptyset$. It satisfies these properties:

Blossoms with the same base vertex have the same base edge.

Suppose $\beta(A)$ is not the base of a maximal blossom, i.e., some blossom B has $A \in \mathcal{A}(B) - \alpha(B)$. Then $\eta(A) \neq \emptyset$ is an edge of $C(B)$.

It is easy to see this notion is well-defined. In particular, in the second property, the definition of blossom B shows the edge of opposite M-type from A exists. Note that $\eta(A) = \emptyset$ only if $\beta(A)$ is the base of a maximal blossom. Using $\emptyset$ as a base edge is handy notation in what follows, e.g., Equation (5.1).

As before, we abbreviate $\beta(B)$ to β when possible, and similarly for $\eta(B)$. *Heavy* and *light* blossoms are defined as before. In Figure 12, the two free loop blossoms have $\emptyset$ base edges. B and B_1 are light blossoms and have base edge η_1. In Figure 5(a), the triangle blossom has η arbitrarily chosen as one of two matched edges incident to its base; similarly, the heavy blossom has η as one of the two unmatched edges incident to its base.

The family of blossoms $\mathcal{A}^*$ constructing B is defined as before. The trails $P_i(v, \beta)$ for f-factor blossoms are the same as before. As before, any trail $P_i(v, \beta(B))$ passes through any $A \in \mathcal{A}^*(B)$ at most once, and if so it traverses a trail $P_j(v, \beta(A))$, $v \in V(A)$ (possibly in the reverse direction). Furthermore, $\eta(A)$ is an edge in $P_i(v, \beta(B))$ unless $\beta(A) = \beta(B)$. To prove this, we can assume A is a maximal blossom with base $\beta(A) \neq \beta(B)$. Let D be the blossom with $A \in \mathcal{A}(D)$ ($D \in \mathcal{A}^*(B) \cup \{B\}$). $P_i(v, \beta(B))$ traverses the blossom trail $C(D)$ on a subtrail, denoted $\overline{P}$ in the proof of Lemma 4.4. For blossom A on $\overline{P}$ with $A \neq \alpha(D)$, $\overline{P}$ either contains both edges of $\delta(A, C(D))$ or, when A is the first vertex of $\overline{P}$, the edge of $\delta(\beta(A), C(D))$ of opposite M-type from A. Both alternatives have $\eta(A)$ in $\overline{P}$.

Mature Blossoms

As before, complementary slackness dictates the sets that may have positive dual variables, and we use this to define mature blossoms. Dual variables are associated with blossom/incident-edge-set pairs (see the review in Appendix B) and the blossoms must satisfy the following "completeness" property: Blossom B with base vertex β and base edge η is *mature* if

every $x \in V(B) - \beta$ has $d(x, M) = f(x)$, and furthermore,
either $d(\beta, M) = f(\beta)$ and η is an edge, or $d(\beta, M) = f(\beta) - 1$ and $\eta = \emptyset$.

We shall see that in contrast to b-matching, the algorithm never creates immature blossoms. Thus, the algorithm does not use a discard step.

Augmenting Trails

Augmenting trails, augmenting blossoms, and the augment step are defined exactly as in b-matching. Any blossom B on the augmenting trail, maximal or not, remains a blossom after rematching. Lemma 4.5 shows this except for exhibiting the base edges of blossoms. We extend that lemma to define base edges for rematched blossoms as follows.

Let AT denote the augmenting trail $(P_0(v, \varepsilon) - (v', \varepsilon)$ in the discussion preceding the lemma). For consistency with the lemma primes denote blossoms after rematching, e.g., B' is the rematched blossom B.

Lemma 5.2. *For any mature blossom B with $\delta(B, AT) \neq \emptyset$, $\eta(B')$ exists and is the unique edge satisfying*

$$\delta(B, AT) - \eta(B) = \{\eta(B')\}.$$

Remark. The lemma applies to all blossoms of our algorithm since all are mature. The lemma also shows that after augmenting, every blossom remains mature—even a free blossom that occurs at an end of AT.

Proof. For some $x \in V(B)$ let xx' be the edge of $\delta(B, AT) - \eta(B)$. To show this edge is uniquely defined, first note the definition of AT implies $d(B, AT)$ is 1 or 2. (An augment may have $d(B, AT) = 0$ but such a B is not mature, the base x has $d(x, M) \leq b(x) - 2$.) If $d(B, AT) = 2$ then $\eta(B) \in E(AT)$, by the definition of augmenting blossom. If $d(B, AT) = 1$, then B contains a free vertex v or v' so $\eta(B) = \emptyset$ by the definition of maturity. In both cases $\delta(B, AT) - \eta(B)$ has exactly one edge.

Next note that AT, a P_i trail, passes through B on a trail $P_j(x, \beta(B))$ (possibly in the reverse direction). Suppose $P_j(x, \beta(B))$ starts with an edge e. Lemma 4.5 shows $\beta(B') = x$ and the M-type of B' is that of the rematched e. This is the opposite of the M-type of the rematched xx'. So we can take $\eta(B') = xx'$.

The remaining possibility is that $P_j(x, \beta(B))$ has no edges, i.e., $j = 0$, $x = \beta(B)$, $B' = B$. There are two possibilities.

Case *B is not free*: xx' alternates with $\eta(B)$. So the rematched xx' has the original M-type of $\eta(B)$. The M-type of B does not change so we can again take $\eta(B') = xx'$.

Case *B is free*: B is a light blossom (even if it is not maximal, by the definition of augmenting blossom). This makes xx' unmatched before rematching ($j = 0$). So the augment makes xx' matched and we can take $\eta(B') = xx'$. □

For any mature blossom B define

$$I(B) = \delta(B, M) \oplus \eta(B). \tag{5.1}$$

The algorithm will assign positive values to dual variables of blossom/incident-edge-set pairs of the form $B, I(B)$ (recall Appendix B). As an example, note this is consistent with ordinary matching and b-matching: Duals are associated with blossoms only because any blossom has $I(B) = \emptyset$. The latter follows since $\eta(B)$ is either the unique matched edge incident to B, or $\emptyset$ when this edge does not exist.

The next lemma will be used to show that an augment step maintains validity of the dual variables (see Lemma 5.7).

Lemma 5.3. *An augment step does not change $I(B)$ for any mature blossom B (maximal or not), i.e, $I(B) = I(B')$.*

make every free atom or blossom an (outer) root of $\overline{\mathcal{S}}$
loop
  if $\exists$ *tight edge* $e = xy \in \overline{E}$ *eligible for* B_x *with* $y \notin \mathcal{S}$ **then**
    `/* grow step */`
    add e, B_y to $\mathcal{S}$
  else if $\exists$ *tight edge* $e = xy \in \overline{E}$ *eligible for both* B_x *and* B_y **then**
    `/* blossom step */`
    if B_x *and* B_y *are in different search trees* **then**
      /* e `plus the` $\overline{\mathcal{S}}$`-paths to` B_x `and` B_y `give an augmenting blossom` B */
      augment M using B and end the search
    $\alpha \leftarrow$ the nca of B_x and B_y in $\overline{\mathcal{S}}$
1   $C \leftarrow$ the cycle $\overline{\mathcal{S}}(\alpha, B_x), e, \overline{\mathcal{S}}(B_y, \alpha)$
    if α *is atomic and* $d(\alpha, M) \leq f(\alpha) - 2$ **then** `/*` α `is a search tree root */`
      augment M using blossom C and end the search
    contract C to an outer blossom with $\eta(C) = \tau(\alpha)$ `/*` C `is the new` B_x `*/`
  else if $\exists$ *inner blossom* B *with* $z(B) = 0$ **then**
    `/* expand step */`
    let $e = \tau(B)$, $f = \eta(B)$, $v = e \cap V(B)$, $\beta(B) = f \cap V(B)$
    let C_e be the subtrail of $C(B)$ traversed by the alternating trail $P_i(v, \beta(B))$,
      where $i \in \{0, 1\}$ is chosen so $P_i(v, \beta(B))$ alternates with e at v
    if $C_e = C(B)$ **then** make B an outer blossom by assigning $\eta(B) \leftarrow e$
2   **else** replace B by C_e `/* the blossoms of` $C - C_e$ `leave` $\mathcal{S}$ `*/`
  else adjust duals

Fig. 13. Pseudocode for an f-factor search.

Proof. Let M' be the augmented matching $M' = M \oplus AT$. Thus

$$\delta(B, M') = \delta(B, M) \oplus \delta(B, AT).$$

Lemma 5.2 shows

$$\{\eta(B')\} = \delta(B, AT) \oplus \eta(B).$$

By definition, $I(B') = \delta(B, M') \oplus \eta(B')$. Substituting the displayed equations transforms this to $\delta(B, M) \oplus \eta(B) = I(B)$. □

5.2 *f*-factor Algorithm

Compared to b-matching, an f-factor algorithm has more restrictions on grow and blossom steps, since edges have a limited number of copies. This necessitates assigning z duals to blossom/incident-edge-set pairs. It also introduces the possibility of matched edges that are not tight. But this simplifies the algorithm by making matched and unmatched edges more symmetric! Similarly, heavy blossoms are required in both the linear programming formulation and the algorithm.

We present the search algorithm as well as the dual adjustment step—the latter differs enough from ordinary matching and b-matching to merit detailed discussion.

The search algorithm is stated in Figure 13. Many notions are defined just as they were for b-matching, specifically the contracted graph $\overline{G}$, its edge set $\overline{E}$, the search structure $\mathcal{S}$, its contraction

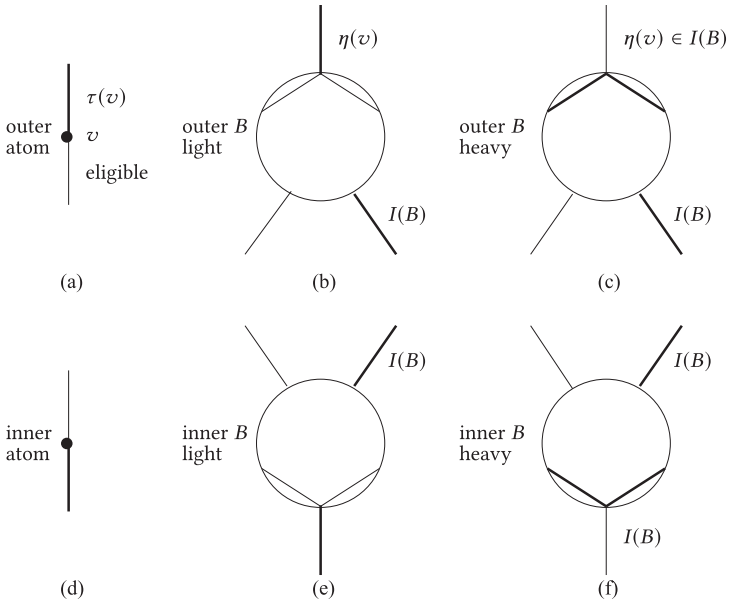

Fig. 14. Six types of nodes v in a search tree. (a)–(c) are outer v, (d)–(f) inner. As labeled in (a), $\tau(v)$ edges are always drawn above v and edges eligible for $B_x = v$ are always below. Edges in $I(B)$ sets are so labeled. In (e) and (f), $\tau(v)$ can be matched or unmatched.

$\overline{\mathcal{S}}$, the B_x sets denoting blossoms or atoms, and free nodes. The following definitions are illustrated in Figure 14.

To define the inner/outer classification let v be a node of $\overline{\mathcal{S}}$. If v is not a search tree root let $\tau(v)$ be the edge to its parent.

Node v of $\overline{\mathcal{S}}$ is *outer* if any of the following conditions holds:
- v is a search tree root;
- v is an atom with $\tau(v) \in M$;
- v is a blossom with $\tau(v) = \eta(v)$.

Otherwise, v is *inner*, i.e., $\tau(v)$ exists but either of the following holds:
- v is an atom with $\tau(v) \notin M$;
- v is a blossom with $\tau(v) \neq \eta(v)$.

In contrast with b-matching, an outer blossom can have $\tau(v) \notin M$ (i.e., when it is heavy) and an inner blossom can have $\tau(v) \in M$. These possibilities are illustrated by B_2 and B_3, respectively, in Figure 12.

Eligibility is defined so that paths in $\overline{\mathcal{S}}$ have the same structure as $C(B)$ trails in blossoms (see Figure 14):

An edge $e = xy \in \overline{E} - \overline{\mathcal{S}}$, is *eligible for* B_x if any of the following conditions holds:
- x is an outer atom and $e \notin M$;
- x is an inner atom and $e \in M$;
- B_x is an outer blossom;
- B_x is an inner blossom and $e = \eta(B_x)$.

$\delta_1 \leftarrow \min\{|\widehat{yz}(e) - w(e)| : e = xy \in E \text{ eligible for } B_x \text{ with } y \notin \mathcal{S}\}$
$\delta_2 = \min\{|\widehat{yz}(e) - w(e)|/2 \ : \ e = xy \in \overline{E} \text{ eligible for both } B_x \text{ and } B_y\}$
$\delta_3 = \min\{z(B)/2 : B \text{ an inner blossom of } \overline{\mathcal{S}}\}$
$\delta = \min\{\delta_1, \delta_2, \delta_3\}$
for *every vertex* $v \in \mathcal{S}$ **do**
  if B_v *is inner* **then** $y(v) \leftarrow y(v) + \delta$ **else** $y(v) \leftarrow y(v) - \delta$
for *every blossom* B *in* $\overline{\mathcal{S}}$ **do**
  if B *is inner* **then** $z(B) \leftarrow z(B) - 2\delta$ **else** $z(B) \leftarrow z(B) + 2\delta$

Fig. 15. Dual adjustment step for f-factors.

In the algorithm statement of Figure 13, the current matching M and the paths $\overline{\mathcal{S}}(x, y)$ are as before. Using the above definitions, much of the algorithm is identical to b-matching.

Now consider the dual adjustment step of Figure 15. We first recall terminology explained in detail in Appendix B. Similar to b-matching, the function $\widehat{yz} : E \to \mathbb{R}$ is defined by

$$\widehat{yz}(e) = y(e) + z\{B : e \in \gamma(B) \cup I(B)\}. \tag{5.2}$$

Say edge e is *dominated, tight,* or *underrated* depending on whether $\widehat{yz}(e)$ is $\geq w(e)$, $= w(e)$, or $\leq w(e)$, respectively; *strictly dominated* and *strictly underrated* correspond to $> w(e)$ and $< w(e)$, respectively. The complementary slackness conditions for optimality require e to be dominated if it is unmatched, as in b-matching. The condition for positive z-duals is that of b-matching extended to include $I(B)$ sets. Finally, a new condition is that e must be underrated if it is matched.

As usual, our algorithm maintains duals to satisfy these requirements. As in b-matching there may be strictly dominated unmatched edges; symmetrically there may be strictly underrated matched edges. (This is expected since our algorithm has minimum cost network flow as a special case.) The absolute values in the definitions of δ_1 and δ_2 reflect these possibilities, as $\widehat{yz}(e) - w(e)$ may have arbtrary sign. The use of $\widehat{yz}(e)$ rather than $y(e)$ (as in ordinary matching and b-matching, Figures 18–19) reflects the possibility that eligible edges can be in $I(B)$ sets and so have positive z contributions in $\widehat{yz}(e)$.

The rest of this section follows the same organization as b-matching, giving clarifying remarks, invariants, examples, and then the formal proof of correctness.

Remarks. Many remarks for b-matching still apply, the exceptions being the simplifications in the blossom and expand steps.

In the blossom step. consider the cycle C, which is constructed in line 1 and then processed as either an augmenting blossom or an outer blossom. When the augment step is executed, we do not consider C a blossom of the algorithm (since it is not mature). When C is processed as an outer blossom, the definition $\eta(C) = \tau(\alpha)$ assumes $\tau(\alpha) = \emptyset$ when α is a root of $\overline{\mathcal{S}}$.

The expand step is simpler than b-matching since all blossoms of $C(B)$ are mature. As in ordinary matching a new blossom in $\overline{\mathcal{S}}$ may have z-value 0.

The algorithm maintains η values in blossom, augment, and expand steps. In line 2 η-values of blossoms in C_e are unchanged, so new blossoms of $\overline{\mathcal{S}}$ may be inner or outer.

As before, when duals are modified, our assumption that the graph has an f-factor guarantees the new duals allow further progress (as shown in the discussion following the proof of Equation (5.6)).

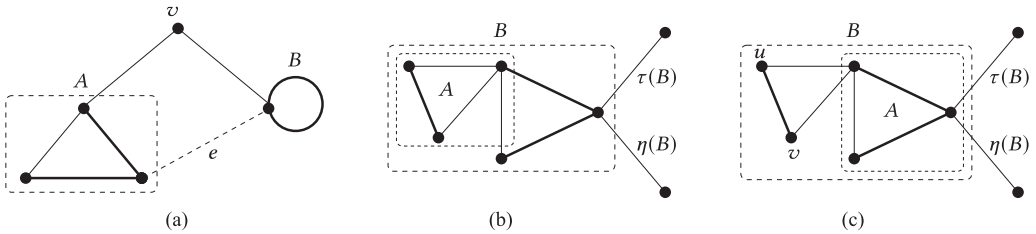

Fig. 16. f-factor algorithm examples.

Invariants. The definition of $\overline{\mathcal{S}}$ *alternating* at node v is unchanged.

(I1) $\overline{\mathcal{S}}$ alternates at any atomic node. Any root blossom is light.
(I2) Every blossom B (maximal or not) is mature.
If B is inner, then it is either a leaf of $\overline{\mathcal{S}}$ or its base edge leads to its unique child.
(I3) For every blossom B (maximal or not) $C(B)$ is a cycle.
(I4) An edge is tight if it is an edge of $\overline{\mathcal{S}}$ or an edge of a contracted blossom. Any nontight edge is dominated if it is unmatched and underrated if matched.

Examples. Strictly Underrated Edges: Figure 12 illustrates how matched edges can become underrated. Suppose a dual adjustment is done. (For this, we assume that α_3 is strictly underrated or that it does not exist.) Since α_5 is incident to an outer atom, it is ineligible, and a dual adjustment decreases $\widehat{yz}(\alpha_5)$ by δ. Since α_2 is incident to an inner blossom and belongs to its I set, it is ineligible and a dual adjustment increases $\widehat{yz}(\alpha_2)$ by $\delta - 2\delta = -\delta$. These adjustments exemplify the two main cases where strictly underrated matched edges are created. (Note also that α_2 or α_5 may be the base edge of a blossom not in $\overline{\mathcal{S}}$. So, in general, even η edges need not be tight.) The underrated edge α_3 may become tight in the dual adjustment since $\widehat{yz}(\alpha_3)$ increases by $(-\delta + 2\delta) + \delta = 2\delta$.

For another example, consider Figure 11(c), assuming the loop e is matched. A dual adjustment decreases $\widehat{yz}(e)$ by 2δ. The same holds if e joins two different free vertices.

Eligibility: Consider Figure 12. No grow step can be done for the ineligible edge α_2. No augment step can be done for α_1 or α_4 since they are ineligible at one end.

Grow Step: In Figure 12, the grow step that added the inner blossom B_3 used a matched edge. This is not possible in b-matching.

Blossom Step: In Figure 16(a), a blossom step can be done for e if $e = \eta(A) = \eta(B)$ and e is tight. Such a blossom step—for two inner nodes joined by an unmatched edge—cannot be done in b-matching. If $e \neq \eta(A)$ then a blossom step cannot be done for e. In this case, e may even complete an augmenting trail (when e is tight and $d(v, M) \leq f(v) - 2$) yet the algorithm does not augment.[10] The same holds if $e \neq \eta(B)$, assuming B remains inner. This situation is further discussed in the Dual Adjustment Step example below.

Augment Step: If an augment step is done for α_3 in Figure 12, the edge joining blossom B_3 to the free root blossom becomes the (unmatched) base edge of both blossoms.

Expand Step: In Figure 16(b), an expand step is done for blossom B. B is made an outer node, with base edge $\tau(B)$. So unlike b-matching an expand step may preserve B as a maximal contracted blossom. If a similar expand step is done for B in Figure 16(c), A is made an inner blossom in $\mathcal{S}$ and

[10]This is the correct strategy: Augmenting might make the current $\eta(A)$ strictly underrated. But it is unmatched.

u and v become atoms not in $\mathcal{S}$. Blossom A may not get expanded in this search even if $z(A) = 0$. If it does get expanded, then A becomes an outer blossom.

Dual Adjustment Step: Returning to Figure 16(a), suppose e is not the base edge of either blossom A, B and B remains inner. A dual adjustment increases $\widehat{yz}(e)$ by $\delta + \delta = 2\delta$. e becomes strictly dominated, and the topologically valid blossom or augment step for e has been destroyed. Subsequent dual adjustments take e even further from being tight. However, this cannot continue forever: Eventually, one of the inner blossoms gets expanded, and dual adjustments now increase $\widehat{yz}(e)$ by $-\delta + \delta = 0$. When the other blossom gets expanded e becomes an unmatched edge joining two outer blossoms/vertices. Now dual adjustments decrease the slack ($\widehat{yz}(e)$ decreases by 2δ) and eventually a blossom step can be done for e.

5.3 Analysis

We follow the same organization as before: The first subsection proves the invariants. The second subsection proves correctness and adds details on initialization and termination. The last subsection proves the desired time bound.

Proof of Invariants

As before, we show all the invariants are preserved by every step of the algorithm. Again our discussion of grow, blossom, augment, and expand steps just treats the most interesting cases. Start by noting that when a search begins any free blossom is light, by (I1) from the previous search. So (I1) is preserved when $\overline{\mathcal{S}}$ is initialized.

Lemma 5.4. *The grow step preserves every invariant.*

Proof. If blossom B_x is inner, then the definition of eligibility implies $e = \eta(B_x)$ and B_y will be the unique child of B_x. So (I2) holds for B_x. e is tight so (I4) continues to hold. □

Lemma 5.5. *The blossom step preserves every invariant.*

Proof. Using the algorithm's notation, $e = xy$ is the edge triggering the blossom step and C is the new outer blossom. We will verify that C satisfies Definitions 4.2 and 5.1, and that C is outer. We first verify the conditions on α, then blossoms $v \in C - \alpha$. There are four possibilities for α.

If α is an outer atom, then (I1) shows all edges to its children are unmatched. Also, if e is incident to α then eligibility implies e is unmatched. So, blossom C is light. This preserves (I1) if α is an atomic search tree root. If α is not a root, then $\tau(\alpha)$ is matched, so it is a valid choice for $\eta(C)$. Contracting C makes $\eta(C) = \tau(C)$ so C is outer.

Similarly, if α is an inner atom, then C is heavy, $\tau(\alpha)$ is unmatched, and $\eta(C) = \tau(C)$ makes C a valid outer blossom.

If α is an outer blossom, then so is C, since $\eta(C) = \eta(\alpha)$.

Finally, α cannot be an inner blossom B: $\eta(B)$ is the only edge of $\delta(B)$ that can lead to a child of B or be e (by (I2) and the definition of eligibility). So, B cannot be the nca of B_x and B_y.

Next, consider a node $v \neq \alpha$ that is a blossom in C. Definition 5.1 requires $\eta(v)$ to be an edge of C. If v is outer this holds since $\eta(v) = \tau(v)$. If v is inner, then, as before, $\tau(v)$ and $\eta(v)$ must be the two edges of $\delta(v, C)$. □

Lemma 5.6. *The expand step preserves every invariant.*

Proof. Case $C_e = C(B)$: We claim α is a vertex. In proof, suppose the contrary. $v \notin \alpha$ makes $P_i(v, \beta(B))$ contain exactly one edge incident to α. $v \in \alpha$ makes $P_i(v, \beta(B))$ contain no edge incident to α. Neither alternative holds in this case.

We conclude vertex α equals $v = \beta(B)$ and $i = 1$ in $P_i(v, \beta(B))$. Thus, e and f are both incident to $\beta(B)$ and both have the same M-type. So changing $\eta(B)$ to e preserves Definition 5.1. Clearly B becomes outer. Since $z(B) = 0$ the change also preserves $\widehat{yz}$ values, i.e., (I4) holds.

Case $C_e \neq C(B)$: To show (I1), let x be an atom in C_e. If $x \neq v, \beta(B)$ then C_e alternates at x. If $x = v$, the choice of i gives the desired alternation. If $x = \beta(B)$, alternation holds since $\eta(B)$ has opposite M-type from B. □

Lemma 5.7. *The augment step preserves every invariant.*

Proof. Recall that an augment step is given a valid augmenting blossom—in particular, a nonatomic end v or v' is a light blossom by (I1). Lemma 5.3 shows no $I(B)$ set changes in the augment. Thus, Equation (5.2) shows every $\widehat{yz}(e)$ value remains the same and (I4) is preserved. □

Lemma 5.8. *A dual adjustment preserves (I4) unless $\delta = \infty$.*

Proof. Any dual adjustment step has $\delta > 0$. We will check (I4) for an arbitrary edge $e = uv$. If $B_u = B_v$ is a blossom, then (as in ordinary matching) $\widehat{yz}(e)$ does not change. So, assume the opposite case, i.e., $B_u \neq B_v$ or e is a loop that is not a blossom (i.e., $u = v$ is an atom). In both cases, $e \in \overline{E}$.

The quantities in $\widehat{yz}(e) = y(e) + z\{B : e \in \gamma(B) \cup I(B)\}$ that may change in a dual adjustment are limited to $y(x)$ and $z(B_x)$ for $x \in \{u, v\}$ (since $z(B)$ changes only on maximal blossoms B). Clearly these quantities do not change if $B_x \notin \overline{\mathcal{S}}$. From now on, assume $x \in \{u, v\}$ with $B_x \in \overline{\mathcal{S}}$. (If e is a loop, consider the possibilities $x = u$ and $x = v$ to be distinct.)

$y(x)$ changes by $\pm\delta$ and, if B_x is a blossom, $z(B_x)$ changes by $\mp 2\delta$, but this contributes to $\widehat{yz}(e)$ iff $e \in I(B_x)$. Define $\Delta(B_x)$ to be the total change in $\widehat{yz}(e)$ at the x end. So $\widehat{yz}(e)$ changes by $\Delta(B_u) + \Delta(B_v)$. (This hold for loops e too.) It is easy to see that $\Delta(B_x) = \pm\delta$, more precisely,

$$\Delta(B_x) = \begin{cases} -\delta & \text{if } B_x \text{ is an outer atom, or an outer blossom with } e \notin I(B_x), \\ & \text{or an inner blossom with } e \in I(B_x) \\ +\delta & \text{if } B_x \text{ is an inner atom, or an outer blossom with } e \in I(B_x), \\ & \text{or an inner blossom with } e \notin I(B_x). \end{cases} \tag{5.3}$$

Define a sign σ by

$$\sigma = \begin{cases} +1 & e \notin M \\ -1 & e \in M. \end{cases}$$

In the following analysis, it may be helpful to consult Figure 14. We consider two main cases.

Case $e \in \overline{\mathcal{S}}$: We claim

$$\Delta(B_x) = \begin{cases} +\sigma\delta & e = \tau(B_x) \\ -\sigma\delta & e \neq \tau(B_x). \end{cases} \tag{5.4}$$

The claim implies $\Delta(B_u) + \Delta(B_v) = 0$ since one of B_u, B_v is the child of the other. Hence, $\widehat{yz}(e)$ does not change and (I4) holds.

To prove the claim, consider the value of $\Delta(B_x)$ in two symmetric cases.

Subcase $\Delta(B_x) = -\delta$: Suppose $e \neq \tau(B_x)$, i.e., e goes to a child of B_x. In all three cases of Equation (5.3), e is unmatched. Thus, $\Delta(B_x) = -\sigma\delta$ as claimed.

Suppose $e = \tau(B_x)$, i.e., e goes to the parent of B_x. In all three cases of Equation (5.3), e is matched. Thus, $\Delta(B_x) = \sigma\delta$ as claimed.

Subcase $\Delta(B_x) = \delta$: Suppose $e \neq \tau(B_x)$, e goes to a child. In all three cases of Equation (5.3), e is matched. Thus, $\Delta(B_x) = -\sigma\delta$ as claimed.

Suppose $e = \tau(B_x)$, e goes to the parent. In all three cases of Equation (5.3), e is unmatched. Thus, $\Delta(B_x) = \sigma\delta$ as claimed.

Case $e \notin \overline{\mathcal{S}}$: Let B_x be a node of $\overline{\mathcal{S}}$. We claim

$$\Delta(B_x) = \begin{cases} -\sigma\delta & e \text{ eligible for } B_x \\ +\sigma\delta & e \text{ ineligible for } B_x. \end{cases} \tag{5.5}$$

The first possibility, e eligible for B_x, is exactly the same as $e \neq \tau(x)$ checked previously. So suppose e is ineligible. As before, consider the value of $\Delta(B_x)$.

Subcase $\Delta(B_x) = -\delta$: This is impossible in the middle case, i.e., B_x outer with $e \notin I(B_x)$, since e is eligible. In the other two cases of Equation (5.3), e is matched. Thus, $\Delta(B_x) = \sigma\delta$ as claimed.

Subcase $\Delta(B_x) = \delta$: This is impossible in the middle case. In the other two cases of Equation (5.3), e is unmatched. Thus, $\Delta(B_x) = \sigma\delta$ as claimed.

Now we show (I4). For every edge e, define

$$slack(e) = \sigma(\widehat{yz}(e) - w(e)).$$

(I4) requires every edge to have nonnegative slack. A dual adjustment changes $slack(e)$ by $\sigma(\Delta(B_u) + \Delta(B_v))$. Equation (5.5) shows $slack(e)$ decreases iff e is eligible for B_x.

At least one of B_u, B_v is a node of $\overline{\mathcal{S}}$, so assume $B_u \in \overline{\mathcal{S}}$. If B_v is also in $\overline{\mathcal{S}}$ we can assume $\Delta(B_u) = \Delta(B_v)$, since, otherwise, $\widehat{yz}(e)$ does not change and the lemma holds. So, in the two cases below, when B_v is a node of $\overline{\mathcal{S}}$ either e is eligible for both B_u and B_v or ineligible for both.

Subcase e *ineligible for* B_u: The dual adjustment increases the slack. Clearly (I4) continues to hold.

Subcase e *eligible for* B_u: When the dual adjustment starts. any edge has nonnegative slack, i.e., $slack(e) = |\widehat{yz}(e) - w(e)|$. If $B_v \notin \overline{\mathcal{S}}$, then initially we have $|\widehat{yz}(e) - w(e)| \geq \delta_1 \geq \delta$. Since $slack(e)$ decreases by δ, $slack(e) \geq 0$ after the dual adjustment and (I4) holds. Similarly, if $B_v \in \overline{\mathcal{S}}$, initially $slack(e) = |\widehat{yz}(e) - w(e)| \geq 2\delta_2 \geq 2\delta$. $slack(e)$ decreases by 2δ so after the dual adjustment $slack(e) \geq 0$ and (I4) holds. □

Reconsidering the last subcase, when $\delta = \delta_1$, the corresponding minimizing edge becomes tight. Thus, a grow step can be done in the next iteration. Similarly, when $\delta = \delta_2$ a blossom step has become possible. Taking δ_3 into account, we see that for any $\delta < \infty$, the dual adjustment step makes at least one grow, blossom, or expand step possible, just like ordinary matching and b-matching.

We have now shown that every invariant is preserved throughout the algorithm.

Termination, Correctness, and Initialization

The algorithm does not loop, by exactly the same argument as b-matching. Next we show the algorithm halts with an f-factor, i.e., no free vertices. We prove this using the fact that the maximum size of a partial f-factor is

$$\min \left\{ f(I) + |\gamma(O)| + \sum_C \left\lfloor \frac{f(C) + |E[C,O]|}{2} \right\rfloor \right\}, \tag{5.6}$$

where the set is formed by letting I and O range over all pairs of disjoint vertex sets, and in the summation C ranges over all connected components of $G - I - O$ [34, Theorem 32.1]. Also, throughout this section, $E[A, B]$, for $A, B \subseteq V$, denotes the set of edges with one end in A and the

other in B. Our derivation provides an alternate proof of this min-max relation. We break the proof up into two claims.

Claim. *Equation (5.6) upper-bounds the size of any partial f-factor.*

Proof. Any edge e of G satisfies exactly one of these conditions:

(i) e is incident to an I vertex;
(ii) e joins two O vertices;
(iii) e joins a vertex in some component C to another vertex of C or to an O-vertex.

Call e type (i), (ii), or (iii) accordingly. We shall see these three types correspond, respectively, to the three terms of Equation (5.6). Note that a loop e may have any of the three types.

Clearly, the number of matched edges of type (i) and (ii) is bounded by the first two terms of Equation (5.6), respectively. For type (iii) consider any component C. Counting edge ends shows the number of matched edges of type (iii), $|E[C, C \cup O] \cap M|$, satisfies

$$2|E[C, C \cup O] \cap M| = \sum_{x \in C} d(x, M) - |E[C, I] \cap M| + |E[C, O] \cap M|. \tag{5.7}$$

Obviously, this implies

$$2|E[C, C \cup O] \cap M| \leq f(C) + |E[C, O]|. \tag{5.8}$$

The third term of Equation (5.6) follows. So, Equation (5.6) is a valid upper bound. □

Claim. *The upper bound of Equation (5.6) is tight.*

Proof. Consider a search that *fails*, i.e., no grow, blossom, or expand step can be done and the dual adjustment step has $\delta = \infty$. Since $\delta_1 = \delta_2 = \infty$, no edge of $E[\overline{\mathcal{S}}, V - S \cup \overline{\mathcal{S}}]$ is eligible at its $\overline{\mathcal{S}}$ ends. Since $\delta_3 = \infty$, there are no inner blossoms.

Let I be the set of inner atoms and O the set of outer atoms. Deleting $I \cup O$ gives a collection of connected components C. There are exactly $f(I)$ matched edges of type (i). This follows since an inner atom is not free, and a matched edge joining two inner atoms is eligible at both ends. There are exactly $|\gamma(O)|$ matched edges of type (ii), since an unmatched edge joining two outer atoms is eligible at both ends. For the type (iii) edges we will prove that any component C has a value $\Delta \in \{0, 1\}$ with

$$\sum_{x \in C} d(x, M) - |E[C, I] \cap M| + |E[C, O] \cap M| = f(C) + |E[C, O]| - \Delta. \tag{5.9}$$

With Equation (5.7) this shows the left-hand side of Equation (5.8) is within 1 of the right. So $|E[C, C \cup O] \cap M| = \lfloor \frac{f(C)+|E[C,O]|}{2} \rfloor$. Hence, the number of type (iii) matched edges equals the third term of Equation (5.6). Thus, Equation (5.6) is a tight upper bound on the size of a partial f-factor.

To prove Equation (5.9), consider two types of components C:

Case $C \subseteq V - \mathcal{S}$: Since no vertex of C is free, the first term on the left of Equation (5.9) is $f(C)$. Consider an edge e from C to a node $v \in \overline{\mathcal{S}}$; e is not eligible for v, so v is not a blossom. Furthermore, $v \in I$ implies $e \notin M$ and $v \in O$ implies $e \in M$. Thus, the second term on the left of Equation (5.9) is 0 and the third term is $|E[C, O]|$. So, $\Delta = 0$ as desired.

Case C *contains an* $\overline{\mathcal{S}}$*-node*: We first show that C is a collection of blossoms forming a subtree of $\overline{\mathcal{S}}$ with no other edges, i.e., $\gamma(C, \overline{E}) \subseteq \overline{\mathcal{S}}$. Any $\overline{\mathcal{S}}$-node B_x of C is an outer blossom ($x \notin I \cup O$). Consider an edge $e = xy \in \overline{E}$ with B_x a blossom of C and B_y a node of C. B_y is also an $\overline{\mathcal{S}}$-node, since e is not eligible for B_x. So B_y is also an outer blossom. e is not eligible for at least one of B_x, B_y, so e is an edge of $\overline{\mathcal{S}}$, as claimed.

Let B_r be the root of the subtree C. Let $e = rs$ be the edge of $\overline{\mathcal{S}}$ from B_r to its parent B_s, if such parent exists; $e = \emptyset$ if B_r is a free blossom. We claim

$$\delta(C, M) \oplus E[C, O] \subseteq \{e\}. \tag{5.10}$$

Take any edge $xy \in \delta(C) - e$, $B_x \in C$. $y \in I \cup O$ (since C is a connected component of $G - I - O$). So the claim Equation (5.10) is equivalent to $xy \in M$ iff $y \in O$. This follows from two cases: If xy is an edge of $\overline{\mathcal{S}}$, then y is a child of B_x. Thus, $xy \in M$ iff y is outer. If xy is not an edge of $\overline{\mathcal{S}}$, then it is not eligible for y. Again, $xy \in M$ iff y is outer.

Now we show Equation (5.9) holds with $\Delta = 1$ in each of three possibilities for e.

If $e = \emptyset$, then B_r contains the unique free vertex of C. Using Equation (5.10), the three terms on the left of Equation (5.9) are $f(C) - 1$, 0 and $|E[C, O]|$.

If e is an edge, then $B_s = s$ is atomic. If s is inner, then $e \in M$. Using Equation (5.10), the three terms on the left of Equation (5.9) are $f(C)$, 1 and $|E[C, O]|$. If s is outer, then $e \notin M$. Using Equation (5.10) the terms are $f(C)$, 0 and $|E[C, O]| - 1$.

We conclude the upper bound (5.6) is tight. □

We have also shown that our algorithm, executed on an arbitrary input graph, halts with a partial f-factor of maximum cardinality. In proof, the analysis of a failed search shows if the algorithm halts because $\delta = \infty$, the current matching has size Equation (5.6), so its cardinality is maximum.

Now we verify that our algorithm is correct, i.e., assuming an f-factor exists the algorithm finds one of maximum weight. We have verified the algorithm's final matching is an f-factor. It remains to verify the LP conditions for optimality (Appendix B). (I4) gives the complementary slackness conditions for matched and unmatched edges. We need only discuss the primal inequalities for blossoms and the corresponding complementary slackness conditions.

The blossom inequalities state that every pair B, I ($B \subseteq V$, $I \subseteq \delta(B)$) satisfies

$$|(\gamma(B) \cup I) \cap M| \leq \left\lfloor \frac{f(B) + |I|}{2} \right\rfloor. \tag{5.11}$$

It is easy to see this holds for any f-factor: The degree constraints imply $2|\gamma(B) \cap M| + |I \cap M| \leq f(B)$. Arithmetic gives $2(|\gamma(B) \cap M| + |I \cap M|) \leq f(B) + |I \cap M| \leq f(B) + |I|$ and integrality gives Equation (5.11).

Complementary slackness requires tightness in Equation (5.11) for every positive $z(B, I(B))$. We will show every final blossom B and its set $I(B)$ satisfy

$$2(|\gamma(B) \cap M| + |I(B) \cap M|) = f(B) + |I(B)| - 1, \tag{5.12}$$

i.e., Equation (5.11) is tight. A light blossom B has $\eta(B) \in M - I(B)$, so counting degrees gives $f(B) = 2|\gamma(B) \cap M| + |I(B) \cap M| + 1$. Since $I(B) \cap M = I(B)$ arithmetic gives Equation (5.12). A heavy blossom B has $\eta(B) \in I(B) - M$, so counting degrees gives $f(B) = 2|\gamma(B) \cap M| + |I(B) \cap M|$. Since $I(B) \cap M = I(B) - \eta(B)$ arithmetic gives Equation (5.12). We conclude the algorithm is correct.

The algorithm can be initialized with any partial f-factor F, collection of blossoms $\mathcal{B}$, and dual functions y, z that satisfy the invariants and definitions:

Every blossom is mature. Every free blossom is light.
Every blossom B has $C(B)$ a cycle.
Every blossom B has $I(B) = \delta(B, F) \oplus \eta(B)$, and z is positive only on pairs $(B, I(B))$.
Every edge of a blossom subgraph is tight. Every nontight edge is dominated (underrated) if it is unmatched (matched), respectively.

As before, the simplest choice is any partial f-factor, no blossoms, $z \equiv 0$, and a function y on vertices with $y(e) \geq w(e)$ ($y(e) \leq w(e)$) for every edge e that is unmatched (matched), respectively. This and other initializations are used in Section 5.4. As with b-matching, appropriate initialization shows that for arbitrary input graphs, our algorithm finds a *maximum cardinality maximum weight f-factor*, i.e., a partial f-factor that has the greatest number of edges possible, and subject to that constraint, has the greatest weight possible. See Appendix B.

Efficiency Analysis

The time to find a maximum f-factor is $O(f(V)(m + n \log n))$. The analysis is essentially identical to b-matching. The biggest difference is intepretation of the parameter m. In the simplest case, every copy of a fixed edge xy has the same weight. Then as in b-matching, m denotes the number of nonparallel edges in the given multigraph G. If G has parallel edges with possibly different weights, the same interpretation of m holds if we assume the copies of xy are given together with their multiplicities and weights, sorted by decreasing weight. This follows since a given search refers to at most two copies of any fixed edge xy (for a blossom step), and a new edge xy is chosen with the greatest weight possible. If G is not given in this required form, we assume a preprocessing step does the sort.

As in b-matching, the algorithms of Figures 13 and 15 use linear time. In the expand step, C_e is computed using the blossom data structure of Appendix C, just like b-matching. For tree-blossom merging, the supporting tree T has minor changes from b-matching. In the definition, an f-factor inner blossom is traversed by a trail $P_i(x, \beta)$, $i \in \{0, 1\}$ ($i = 0$ for b-matching). Analogous to b-matching, we define $P_i^-(x, \beta)$ to be $P_i(x, \beta)$ with every maximal subtrail of the form $P_1(\beta(A), \beta(A))$ replaced by the vertex $\beta(A)$. In the new Definition 3.1, an inner blossom B is represented by $P_i^-(x, \beta)$ in T_B. (This trail may actually be the single vertex $\beta(B)$, as illustrated in Figure 16(b) and (c). In the latter, note that $P_i(x, \beta)$ does not pass through u or v.)

The supporting tree is maintained similar to b-matching. A minor difference occurs when line 2 adds a blossom A represented by $\beta(A)$ to $\overline{\mathcal{S}}$. If this makes A outer, then, as before, the remaining vertices of $A - \beta(A)$ are added to the supporting tree. If this makes A inner, the supporting tree does not change. A may become outer in a subsequent expand step (when $z(A) = 0$) or in a blossom step. In both cases, the vertices of $A - \beta(A)$ are added as before.

The tree-blossom-merging algorithm is used to track blossom steps for both unmatched and matched edges. No modifications are needed. To justify this observe that Equation (5.5) shows once xy becomes eligible at both ends, every dual adjustment decreases its slack by 2δ. Thus, the numerical key used in the blossom-tree merging algorithm is correct, i.e., it gives the value of total dual adjustment when xy becomes tight.

This argument depends on the simple fact that once an edge becomes eligible it remains so. This property was also used in ordinary matching and b-matching, but we give a formal proof for f-factors here. Say that an edge uv is *eligible at u* if it is eligible for B_u. The new term avoids referring to the time-varying B_u. This term and the next lemma come in handy in Appendix D.

Lemma 5.9. *Once an edge $e = uv$ becomes eligible at u it remains so, until it leaves $\overline{E} - \overline{\mathcal{S}}$.*

Proof. We claim that if e is eligible at u at some instant, it remains eligible at u as long as it belongs to $\delta(B_u) - \overline{\mathcal{S}}$. Clearly this claim implies the lemma. To prove the claim, we consider three possilities for B_u at the instant e is eligible at u.

Case *B_u is an outer blossom*: B_u remains an outer blossom, even as it gets enlarged in blossom steps. Clearly e remains eligible at u until it enters B_u or becomes an $\overline{\mathcal{S}}$-edge.

Case B_u *is an atom*: B_u can only change by becoming an outer blossom B'_u. If $e \in \overline{E} - \overline{\mathcal{S}}$, it is eligible for B'_u, and the previous case applies.

Case B_u *is an inner blossom*: e must be the edge $\eta(B_u)$. B_u can change by becoming an outer blossom, and again the first case applies. The other possible change occurs if B_u gets expanded. Vertex u remains in $\mathcal{S}$. Let B'_u be the new maximal blossom containing u, $B'_u \in \overline{\mathcal{S}}$. If B'_u is an outer blossom the first case applies. If B'_u is an inner blossom, e is $\eta(B'_u)$. So it remains eligible and this case continues to apply. Finally, suppose B'_u is an atom. The P_i trail used to expand B_u is alternating, including an alternation at u. So, either u is outer and $uv \notin M$ or u is inner and $uv \in M$. In both cases, uv is eligible at u and the previous case applies. □

Theorem 5.10. *A maximum f-factor can be found in time $O(f(V)(m + n \log n))$.*

5.4 Related Algorithms

Degree-Bounded Subgraphs

For two functions $\ell, h : V \to \mathbb{Z}_+$, a subgraph H of G is an (ℓ, h)-*subgraph* if its degree function d_H satisfies $\ell \le d_H \le h$.

We convert such a subgraph into an f-factor as follows. Starting with the given graph G form graph G_s by adding a vertex s, with edges vs of multiplicity $h(v) - \ell(v)$, $v \in V$, and the loop ss of multiplicity $\lfloor h(V)/2 \rfloor$. Every new edge weighs 0. Define a degree requirement function f by

$$f(v) = \begin{cases} h(v) & v \in V \\ h(V) & v = s. \end{cases}$$

The (ℓ, h)-subgraphs H of G correspond to the f-factors F of G_s, and corresponding subgraphs have the same weight. In proof, starting with an H, construct F by adding $h(v) - d_H(v)$ copies of vs for every vertex $v \in V$, and $|E(H)|$ copies of ss. This gives s degree exactly $(h(V) - 2|E(H)|) + 2|E(H)| = h(V) = f(s)$. Obviously, every $v \in V$ has degree $f(v)$, and $w(H) = w(F)$. Similarly, starting with an F, let $H = F - s$. Clearly, H is an (ℓ, h)-subgraph and $w(H) = w(F)$.

Corollary 5.11. *A maximum or minimum weight (ℓ, h)-subgraph can be found in time $O(h(V)(m + n \log n))$. For a minimum weight (ℓ, h)-subgraph the bound improves to $O(\ell(V)(m + n \log n))$ if the weight function is nonnegative or if $h \equiv d_G$ (i.e., we seek a minimum weight ℓ-edge cover [34, chap. 34]).*

Proof. To achieve the first time bound, execute the f-factor algorithm on G_s, using the given weight function w for maximization and $-w$ for minimization. Since $f(V + s) = O(h(V))$ and G_s has $O(m)$ distinct edges, Theorem 5.10 gives the desired bound.

Next, consider minimum weight (ℓ, h)-subgraphs with nonnegative w. Use the f-factor algorithm on G_s with weight function $-w$, and initial dual functions $y \equiv 0$ and $z \equiv 0$ with no blossoms. These duals are feasible for $-w$. To define the initial matching, let $\delta = \ell(V) \bmod 2$. Match every copy of every edge vs, $v \in V$, and $(\ell(V) - \delta)/2$ copies of ss. To show this matching is valid, first note the degree of s in the matching is $(h(V) - \ell(V)) + (\ell(V) - \delta) = h(V) - \delta \le f(s)$. Also, every matched edge is tight since $y \equiv 0$.

The number of searches of the f-factor algorithm is $(\ell(V) + \delta)/2 = O(\ell(V))$. The time bound for nonnegative w follows.

Finally, suppose w is arbitrary but $h \equiv d_G$. Let $N = \{e : w(e) < 0\}$ and let G' be the graph $G - N$. The minimum weight (ℓ, h)-subgraph consists of N plus a minimum weight (ℓ', h')-subgraph on G', where $\ell' \equiv \max\{\ell - d_N, 0\}$ and $h' \equiv d_{G'}$. Since G' has a nonnegative weight function, the previous case shows the time is $O(\ell(V)(m + n \log n))$. □

Strongly Polynomial Algorithm

We use essentially the same reduction to bipartite matching as b-matching. Assume the multigraph G is specified by a function $c : V \times V \to \mathbb{Z}_+$ that gives the number of parallel copies of each edge. The algorithm below rounds c up to ensure that edges do not disappear.

Define graph G' by setting $f' = 2\lfloor f/2 \rfloor$ and $c' = 2\lceil c/2 \rceil$. Let M' be a maximum cardinality maximum weight f'-factor on G' with corresponding optimal dual function y. For every edge e with $c'(e) > c(e)$ copies of e in M', remove 1 copy of e from M'. Let M be the resulting partial f-factor on G. Using M, y (and $z \equiv 0,\ \mathcal{B} = \emptyset$) as the initial solution, execute the f-factor algorithm of Section 5.2 on G.

The analysis is similar to b-matching. Since we assume G has an f-factor, G' has a partial f'-factor with $\geq \frac{f(V)}{2} - n$ edges. At most m matched edges are deleted to form M. So our f-factor algorithm performs $\leq m + n$ augmentations. Thus the time for the entire algorithm is $O(m(m + n \log n))$ plus the time to find M', y. As before, the latter strictly dominates the time.

We find M', y using a graph G^+ similar to b-matching: Extend graph G' to G^+ by adding a vertex s with degree constraint

$$f'(s) = f'(V),$$

and edges vs ($v \in V$) and ss with multiplicities and weights given, respectively, by

$$c'(vs) = f'(v),\ c'(ss) = f'(V),\ w(vs) = 0,\ w(ss) = Wf'(s) \text{ for } W = \max\{1, |w(e)| : e \in E(G)\}.$$

Note that f' and c' remain even-valued functions. A maximum cardinality maximum weight f'-factor of G' corresponds to a maximum f'-factor of G^+. The proof is exactly the same as b-matching.

As before, we find a maximum f'-factor of G^+ by reducing to a bipartite graph BG. BG has vertices v_1, v_2 ($v \in V(G^+)$) and edges u_1v_2, v_1u_2 ($uv \in E(G^+)$) with degree constraints, multiplicities, and edge weights given, respectively, by

$$f'(v_1) = f'(v_2) = f'(v)/2,\ c'(u_1v_2) = c'(v_1u_2) = c'(uv)/2,\ w(u_1v_2) = w(v_1u_2) = w(uv).$$

A loop uu of G^+ gives edge u_1u_2 with multiplicity $c'(uu)$ in BG. Let x be a maximum f'-factor on BG with optimum dual function y. Define an f'-factor M' on G^+ by taking $x\{u_1v_2, u_2v_1\}$ copies of each edge $uv \in E(G^+)$ (by our summing convention this means $x(u_1u_2)$ copies of a loop uu). x exists and M' is a maximum f'-factor on G^+, by the same proof as b-matching. (In applying the Euler tour technique to prove (b), start the Euler tour with a closed trail of $2\lfloor x(uv)/2 \rfloor$ copies of uv for every $uv \in E(G^+)$. This includes loops uu.) Define a dual function y by $y(v) = y\{v_1, v_2\}/2$. Applying complementary slackness to the definition of the dual function for bipartite f-factors [34, chap. 21] we get that an f'-factor x on BG and a dual function y are both optimum iff for every edge e of BG,

$$x(e) = 0 \Longrightarrow y(e) \geq w(e);\ x(e) = 1 \Longrightarrow y(e) \leq w(e). \tag{5.13}$$

Now consider an edge $e = uv$ of G^+ (e may be a loop). The matching x on BG has a mirror image x' defined by $x'(a_1b_2) = x(b_1a_2)$. Suppose some copy of e in BG is unmatched, say $x(u_1v_2) = 0$. (5.13) implies $y(u_1v_2) \geq w(uv)$ as well as $y(v_1u_2) \geq w(uv)$. Thus,

$$y(e) = \Big((y(u_1) + y(u_2)) + (y(v_1) + y(v_2))\Big)/2 \geq 2w(uv)/2 = w(e).$$

Similarly, if some copy of e in BG is matched then $y(e) \leq w(e)$. So the functions $y, 0$ are optimum duals for an f'-factor on (the non-bipartite graph) G^+.

The matching M defined from M' is clearly valid on G (i.e., nonexistent matched edges are deleted). y is also optimum on G (an unmatched edge of G is present in G^+ since c' rounds up). We

conclude that restricting M and y to G, along with $z \equiv 0$, $\mathcal{B} = \emptyset$, gives permissible initial values for our f-factor algorithm. In conclusion the main algorithm is correct.

The problem on BG is a capacitated transportation problem, where x is an optimum integral solution and y is an optimum dual function [34, chap. 21]. We solve it using Orlin's algorithm [29]. It reduces the capacitated transportation problem to the uncapacitated case. The reduction modifies the graph, but it is easy to see that the optimum dual function y on the modified graph gives an optimum dual on the given graph. (Alternatively, an optimum dual function can be found from x itself using a shortest path computation, in time $O(nm)$ [2].)

Orlin solves the capacitated transportation problem (more generally, capacitated transhipment) in time $O((m \log n)(m + n \log n))$ [29]. It gives both x and y. Using this, we obtain our strongly polynomial bound:

Theorem 5.12. *A maximum f-factor can be found in time $O(\min\{f(V), m \log n\}(m + n \log n))$.*

T-joins

Recall that for any set of vertices T of even cardinality, a *T-join* is a subgraph of G that has T as its set of odd-degree vertices. For any edge-cost function, c, it is of interest to find a minimum cost T-join. We proceed as follows.

Let N be the set of edges of negative cost. Define $t = |T| + 2|N|$. Let G' be the graph G enlarged by adding $t/2$ loops at every vertex, where each loop has cost 0. Define a degree-constraint function

$$f(v) = \begin{cases} t - 1 & v \in T, \\ t & v \notin T. \end{cases}$$

A minimum cost f-factor is a minimum cost T-join augmented by enough loops to satisfy the degree constraints exactly. In proof, let J be a minimum T-join and F a minimum f-factor. $c(F) \geq c(J)$ since F with all loops deleted gives a T-join of the same cost.

For the the opposite inequality, note that *wlog* J consists of $|T|/2$ paths, each joining two vertices of T, and $\leq |N|$ cycles (since we can assume every cycle contains a negative edge). Thus, any vertex has degree $d(v, J) \leq |T| + 2|N| = t$, with strict inequality if v is a terminal. Furthermore, $d(v, J)$ and $f(v)$ have the same parity. Hence we can add loops at each vertex to make J an f-factor. We conclude $c(J) \geq c(F)$.

For our algorithm, define edge weights to be the negatives of edge costs. So we seek a maximum weight f-factor of G'. Initialize the algorithm with a matching M consisting of every N-edge, and enough loops at each vertex to make $f(v) \geq d(v, M) \geq f(v) - 1$. Furthermore, $y \equiv 0$ and there are no blossoms. (This initialization is valid since every loop is tight and for edges of G, every matched edge is underrated and every other edge is dominated.) Then execute the f-factor algorithm.

The f-factor algorithm performs $\leq n/2$ searches. A search uses time $O(m + n \log n)$ – although the graph has many loops, only two loops at each vertex are processed in any given search. Also note these special cases: When there are no negative edges, there are $|T|/2$ searches. When there are no terminals, there are $\leq |N|$ searches.

Theorem 5.13. *A minimum cost T-join can be found in time $O(n(m + n \log n))$. If costs are nonnegative the time is $O(|T|(m + n \log n))$. If there are no terminals the time is $O(\min\{|N|, n\}(m + n \log n))$.*

Shortest Paths

Consider the shortest-path problem on a connected undirected graph G with a conservative cost function c, i.e., negative edge costs are allowed but any cycle has nonnegative cost. We are interested in the single-source shortest-paths problem, i.e., given a source vertex s, we wish to find a shortest path from each vertex v to s. We will show the blossom tree provides a generalized

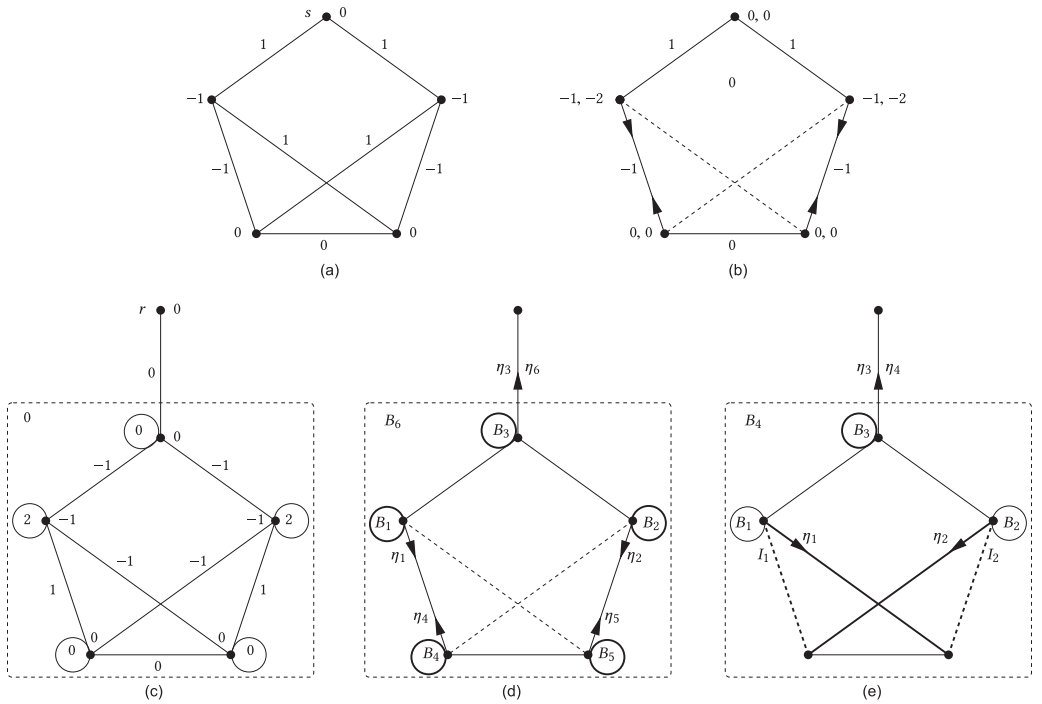

Fig. 17. Shortest-path algorithm. (a) The given graph G with edge costs and distances from s. (b) The gsp-structure. (c) G_r when $\delta = \infty$: Edge weights and final dual values. (d) Matching of all loops, with its blossoms B_i. (e) Arbitrary matching with its blossoms B_i. Each B_i has base edge η_i. Every edge is tight. In (e), the I_i edges are not in a blossom subgraph.

shortest-paths tree, in four steps. Section 5.4.1 gives a "base algorithm." It preprocesses the graph to quickly return the shortest path for a given vertex. The base algorithm works for simple edge cost functions. (In fact, it works for a slight perturbation of *any* conservative cost function.) Section 5.4.2 shows the base algorithm gives a succinct representation of all shortest-paths from a given source s. Also, for readers familiar with the generalized shortest-paths tree (the "gsp structure") introduced by Gabow and Sankowski [19], it verifies that our representation is precisely that structure. Section 5.4.3 extends the base algorithm to work for arbitrary conservative costs. Section 5.4.4 shows the time bound for the entire algorithm is $O(n(m + n \log n))$, the best-known time bound to find a shortest sv-path for two given vertices s, v. A more precise bound based on the number of negative edges is also given.

Figure 17 illustrates the discussion. Part (a) gives a conservative graph, with distances labelling each vertex. Part (b) is the gsp-representation. (The arrow from a vertex v gives the first edge of its shortest path, and the path continues in that direction. Further details are in Section 5.4.2.) Our algorithm constructs the gsp-representation using an f-factor on the graph of part (c). Notice how the optimum f-factor of part (d) resembles the gsp-representation. Less so for the optimum f-factor of part (e). Part (d) illustrates the base algorithm, part (e) is the general case.

5.4.1 The Base Algorithm. We will use the f-factor algorithm to find a search structure $\overline{\mathcal{S}}$ that handles shortest-path queries. Specifically for any vertex v, a shortest vs-path P will be composed of P_i trails in $\overline{\mathcal{S}}$. The query algorithm finds P in time proportional to its length. The base algorithm

accomplishes this when every cycle of the given graph has positive cost. (A more general criterion is given in what follows.)

The following *base algorithm* consists of a preprocessing step that constructs $\overline{\mathcal{S}}$ and a query algorithm that returns a shortest vs-path for a given v.

Preprocessing. Define edge weights as the negatives of edge costs. Let G' be the graph G with a loop of weight 0 added at every vertex and the degree-constraint function $f(v) = 2$ for every $v \in V$. Execute the f-factor algorithm to find a maximum f-factor M and corresponding duals.

Let G_r be the graph G' enlarged by a new vertex r with degree-constraint $f(r) = 1$ and new weight 0 edge rs (Figure 17(c)). Perform a search of the f-factor algorithm: Initialize the search with matching M, its duals, and $y(r) = -y(s)$. Halt when $\delta = \infty$. M remains the matching but the duals and blossoms may change. Let $\overline{\mathcal{S}}$ be the final search structure and y the final dual function.

Query algorithm. Given a vertex v, return a shortest vs-path, as follows. Return the quantity $y(v) - y(s)$ as the shortest distance from v to s. Let P be the path in $\overline{\mathcal{S}}$ from B_v to B_s. Let $\overline{P}$ consist of P plus for each blossom B in P, the trail $P_1(x, \beta(B))$ where

$$x = \begin{cases} v & B = B_v \\ \eta(A) \cap V(B) & A \text{ the blossom preceding } B \text{ in } P. \end{cases}$$

Return the trail $\overline{P}^-$ consisting of $\overline{P}$ with its loop edges deleted.

Analysis. We prove the base algorithm is correct for arbitrary conservative costs, *assuming* the matching M returned by our algorithm consists of all the loops. (This assumption includes the fact that M has been found by our f-factor algorithm, so its corresponding blossom structure is also available.) The assumption holds if every cycle of G has positive cost. But it needn't hold in general (e.g., Figure 17(e)).

Proposition 5.14. *$\mathcal{S}$ contains every vertex of G_r. Every node of $\overline{\mathcal{S}}$ is outer. r is an atom and every other node is a blossom. Every blossom, maximal or not, is heavy.*

Proof. There are no inner blossoms since $\delta_3 = \infty$. There are no inner atoms x since the matched loop xx would make $\delta_2 < \infty$. $\mathcal{S}$ contains every vertex since G_r is connected, and an edge from an outer node to a non-$\overline{\mathcal{S}}$ node would make $\delta_1 < \infty$.

r is an atom since it has degree one. There are no other outer atoms since M consists of loops. No blossom B, maximal or not, is light, since a light blossom has $\eta(B)$ matched and not a loop. □

To analyze the query algorithm fix a query vertex v. Consider the definition of $\overline{P}^-$. For each x, $P_1(x, \beta(B))$ is an alternating trail that starts and ends with a matched edge (since B is heavy). So the edges are alternately matched loops and edges of E. $\overline{P}^-$ is a path, i.e., no repeated vertices. This follows since a repeated vertex in a $P_1(x, \beta(B))$ trail comes from a subtrail $P_1(\beta(B'), \beta(B'))$. This subtrail is the loop $(\beta(B'), \beta(B'))$, which gets discarded from $\overline{P}$.

To show $\overline{P}^-$ is a shortest path define the graph G_v to be G_r enlarged with a vertex v' that has $f(v') = 1$ and a weight 0 edge vv'. Set $y(v') = -y(v)$. G_v satisfies all the invariants of the f-factor algorithm, so we can imagine a hypothetical search of that algorithm. The hypothetical search executes just one step, a blossom step for edge vv', which augments the matching. This follows since vv' is tight and eligible for the outer nodes $B_v, B_{v'}$.

The augmenting trail is $\overline{P} \cup \{rs, vv'\}$. The augmented matching weighs $w(\overline{P}^-)$, since all loops weigh 0 as do the edges rs and vv'. The construction of G_v shows a maximum f-factor has weight equal to $-d(v)$, the negative of the distance from v to s. (Recall G is conservative, so G_v has no positive cycles.) Thus $w(\overline{P}^-) = -d(v)$, i.e., $\overline{P}^-$ is a shortest vs path.

As a last step, we verify that the query algorithm returns the correct shortest-path distance $d(v)$. The key is the next proposition. It actually holds for arbitrary augmenting trails in the f-factor algorithm, generalizing a property of Edmonds' algorithm. For notational simplicity, we only prove the special case needed for our algorithm.

Proposition 5.15. *The augmentation of the hypothetical algorithm changes the weight of the matching by* $y(r) + y(v')$.

Proof. Every edge of the augmenting trail A is tight by (I4). Thus, the weight of the matching changes by $\widehat{yz}(A - M) - \widehat{yz}(A \cap M)$. The y terms make a net contribution of $y(r) + y(v')$, since A alternates at every interior vertex. So, the lemma follows if we show the z terms make no net contribution. To prove this, consider any blossom B, maximal or not. Let AZ be the set of all edges with a $z(B)$ contribution, i.e., $AZ = A \cap (\gamma(B) \cup I(B))$. We claim AZ is an alternating trail of even length. Clearly this implies the $z(B)$ terms make a net contribution of 0 as desired.

To prove the claim, first observe that $I(B) = \{\eta(B)\}$. This follows since B is heavy and M consists of loops. If B is maximal, using the notation of the algorithm gives $AZ = P_1(x, \beta(B)) \cup \eta(B)$. This is an even-length alternating trail as desired. If B is not maximal, recalling the definition of P_i trails shows that exactly the same characterization applies. □

Since all loops weigh 0 the augment changes the weight of the f-factor by $w(\overline{P}^-)$. So the optimum f-factor weighs $y(r) + y(v')$. Thus, $d(v) = -y(v') - y(r) = y(v) - y(r)$. This is the query algorithm's formula, assuming $y(r) = y(s)$. To prove that, observe that in G_s the augmenting trail A weighs $y(r) + y(s')$. $A = (s', s, s, r)$, so it weighs $0 - 0 + 0 = 0$. Thus $0 = y(r) + y(s') = y(r) - y(s)$ as desired.

We note some further properties of our shortest paths, that will be required for the next section. Every loop vv is a blossom. In proof, let B be the smallest blossom containing v. If $v \neq \beta(B)$, then v is atomic in B. $C(B)$ alternates at v so vv is an edge of $C(B)$. But this contradicts invariant (I3). So, $v = \beta(B)$, and B heavy implies B is the loop blossom vv. We have also deduced that vv is tight.

Finally, note that $\overline{P}^-$ is easily specified by using edges $\eta(A)$ to leave blossoms A. By definition, each edge of $\overline{\mathcal{S}}$ is traversed by leaving a blossom A along $\eta(A)$. To traverse the portion of $\overline{P}^-$ in a blossom B, starting at a subblossom A we leave A on $\eta(A)$ and continue along edges of $C(B)$ to reach $\alpha(B)$; each subblossom B' along this route is traversed recursively. As an example, in Figure 17(d) the shortest path from the B_1 vertex starts with edge $\eta_1 = \eta(B_1)$ and continues around the cycle to s. This involves traversing some η edges backwards, e.g., η_4, but this does not violate our description.

5.4.2 The Generalized Shortest-Paths Tree. As mentioned the, final search structure $\mathcal{S}$ is a succinct representation of all shortest paths from a fixed source s. This section sketches how $\mathcal{S}$ corresponds to the gsp-structure.

Our representation amounts to search structure $\mathcal{S}$ of the basic algorithm with r and the matched loops removed. For the simplest example, observe that when every edge cost is positive the edges $\eta(\{vv\}), v \in V - s$ form a shortest-paths tree. In proof, first note that no edge uv is both $\eta(\{uu\})$ and $\eta(\{vv\})$. (Such an edge must have nonnegative weight, since it satisfies $y(u) - w(uv) \leq y(v)$ and $y(v) - w(uv) \leq y(u)$.) This implies any blossom B contains a unique edge xy that is neither $\eta(\{xx\})$ nor $\eta(\{yy\})$ (since $C(B)$ has $|C(B)|$ edges that are $|C(B)| - 1$ η-values). No shortest path uses xy. (The shortest path for x leaves every subblossom $A \in C(B)$ along $\eta(A)$, as shown previously. So $\eta(A)$ points away from xy.) Thus discarding xy from every blossom (which discards loop blossoms too!) gives a shortest-paths tree.

In general, our representation consists of two parts: The overall structure is $\overline{\mathcal{S}}$, a tree whose nodes are contracted blossoms that collectively contain all the vertices of G. The rest of the structure is the set of blossoms, represented as a collection of cycles, corresponding to Definitions 4.2 and 5.1 of blossom and the blossom tree $T(\mathcal{B})$. The base edges of these heavy blossoms serve as pointers to follow shortest paths, in the sense described at the end of last section. Again this is illustrated in Figure 17(d), where there is just one nonsingleton blossom, and the η edges of the heavy loop blossoms define the shortest paths.

The gsp-structure of Gabow and Sankowski [19] also consists of an overall tree whose nodes are trees of nested cycles. Our η pointers are called $\tau(N)$ edges for blossoms N in Reference [19]; the base vertex of N is called t_N. This correspondence in Figure 17 between parts (b) and (d) is clear.

In addition, the gsp-structure has numeric labels that prove its validity. (They guarantee that the paths given by the structure, i.e., the paths of the query algorithm, are, in fact, shortest paths. They also guarantee that the graph is conservative.) To describe the labels in our terminology, first recall that every edge of G is dominated, and is tight if in $\mathcal{S}$. Also, every loop is tight. Furthermore, $d(v) = y(v) - y(s)$. This gives

$$d(u) + d(v) + c(uv) = y(u) + y(v) - 2y(s) - w(uv) \geq -z\{B : uv \in \gamma(B) \cup \eta(B)\} - 2y(s),$$

with equality on $\mathcal{S}$. Define $z' : 2^V \to \mathbb{R}$ by $z'(B) = -z(B)$, $z'(V) = -2y(s)$ to get

$$d(u) + d(v) + c(uv) \geq z'\{B : uv \in \gamma(B) \cup \eta(B)\},$$

with equality for every edge of the representation (and equality for all loops). This is the exact relation satisfied by the labels of the gsp-structure, wherein d labels each vertex, z' labels each node as well as V, and it is required that $z'(B) \leq 0$ for every $B \neq V$. These labels are shown in Figure 17(b); clearly they correspond to our algorithm's numeric labels shown in Figure 17(c).

5.4.3 Arbitrary Conservative Costs. We extend the representation to arbitrary conservative costs by perturbing the costs to eliminate zero-weight cycles. We do this twice. First, we wish to show our representation exists for arbitrary real-valued conservative costs. This is accomplished by a symbolic execution of the base algorithm. Details are in Appendix E. The second goal is an efficient algorithm for integral costs. The details, that preserve both efficiency and integrality, are as follows.

The given cost function c is conservative and integral-valued. We will execute the base algorithm using an integral blow-up of c, specifically the cost function

$$c' = 4nc + 1.$$

c' has no 0-cost cycles. For any v, a shortest sv-path w.r.t. c' is a shortest sv-path w.r.t. c, that, in addition, has the smallest length possible. We execute the base algorithm using c'. Clearly this algorithm can answer shortest path queries for c. However, we wish to find the complete gsp-representation, which uses the optimum dual functions y, z for its numerical labels. (As previously mentioned, they allow correctness of the representation to be verified, in linear time.) So we need another step, to transform the duals given by the algorithm for c' to those for c. The transformed duals must maintain the algorithm's blossom structure.

We use the following terminology. For any duals y, z, and corresponding blossom structure, define the function Z on blossoms B by

$$Z(B) = z\{A : A \supseteq B,\ A \text{ a blossom}\}.$$

Z uniquely defines the dual function z via the relations $z(B) = Z(B) - Z(p(B))$ for $p(B)$ the parent of B in the blossom tree, with the convention $Z(p(B)) = 0$ for every maximal blossom B.

Call any edge e a *witness* for blossom B if $e \in C(B)$ and $e \notin I(A)$ for any blossom $A \subset B$. Since any edge $e \in C(B)$ is tight, i.e., $w(e) = \widehat{y}\widehat{z}(e) = y(e) + z\{A : e \in \gamma(A) \cup I(A)\}$, a witness has the key property $Z(B) = w(e) - y(e)$.

In general, a blossom B may not have a witness.[11] But a blossom at the end of our algorithm does have a witness. In proof, first note this is clear if $C(B)$ is a loop. So assume $C(B)$ contains r nodes, $r > 1$. Every node A is a heavy blossom (Proposition 5.14) and has $I(A) = \{\eta(A)\}$ (since the matching consists of loops). Node $\alpha(B)$ does not have its base edge in $C(B)$. So the r nodes collectively have at most $r - 1$ base edges in $C(B)$. $C(B)$ has r edges that are not loops. So at least one of these edges, say e, is not the base of either of its ends. Thus $e \notin I(A)$ for any blossom A. e is the claimed witness for B.

Let y', z' be the algorithm's duals for c'. Let y, z denote the desired transformed duals for c. To construct y, z first do an extra dual adjustment step to make $y'(s)$ a multiple of $4n$. (In other words use $\delta = y'(s) - 4n\lfloor y'(s)/4n \rfloor$ in a dual adjustment step. Note every vertex is outer for this dual adjustment. For convenience let y', z', Z' now denote these adjusted duals.) The base algorithm provides the shortest c'-path for v, call it P_v. Define the transformed duals by

$$\begin{aligned} y(v) &= -w(P_v) + y'(s)/4n && \text{for every vertex } v \in V \\ Z(B) &= w(e) - y(e) && \text{for every blossom } B \text{ and } e \text{ a witness for } B. \end{aligned}$$

The transformed z is the dual function corresponding to Z. (Note the transformed duals are easily constructed in linear time. In particular $w(P_v) = -c(P_v)$ where $c(P_v) = 4n\lfloor c'(P_v)/4n \rfloor$.)

Lemma 5.16. *The above functions y, z are valid optimum duals. Specifically, they satisfy invariant (I4) for the unmodified weight function $w = -c$ and the f-factor M and blossoms found by the algorithm.*

Proof. Let w' denote the modified weight function used in the algorithm, i.e.,

$$w'(e) = 4nw(e) - \mu(e), \text{ where } \mu(e) = \begin{cases} 1 & e \text{ an edge of } G \\ 0 & e \text{ a loop.} \end{cases}$$

(Recall $w(e) = 0$ for e a loop.) Let $\ell(P_v)$ be the length of the sv-path P_v. Recalling the query algorithm, $y'(v) = -w'(P_v) + y'(s) = -4nw(P_v) + \ell(P_v) + y'(s)$. Thus,

$$y'(v) = 4ny(v) + \ell(P_v).$$

Consider any blossom B (even a loop) and any witness $e = uv \in C(B)$. So,

$$\begin{aligned} Z'(B) &= w'(e) - y'(e) = (4nw(e) - \mu(e)) - 4ny(e) - \ell(P_u) - \ell(P_v) \\ &= 4nZ(B) - r(B), \end{aligned} \tag{5.14}$$

where

$$0 \le r(B) \le 1 + 2(n-1) = 2n - 1.$$

There may be several choices for e, but Equation (5.14) shows $Z(B)$ is uniquely defined, since the interval $[Z'(B), Z'(B) + 2n - 1]$ contains a unique multiple of $4n$.

Observe that $z(B)$ is nonnegative:

$$\begin{aligned} 4nz(B) &= 4n(Z(B) - Z(p(B))) = Z'(B) + r(B) - Z'(p(B)) - r(p(B)) \\ &= z'(B) + r(B) - r(p(B)) \ge -(2n-1). \end{aligned}$$

[11] For instance, if $C(B)$ consists of matched edges, $\alpha(B)$ contributes two I edges. The remaining $|C(B)| - 2$ edges may be I values of the remaining $|C(B)| - 1$ light blossoms.

The last inequality follows since $z'(B) \geq 0$ and $r(p(B)) \leq 2n-1$ (even if B is maximal). Clearly, $4nz(B) \geq -(2n-1)$ implies $z(B) \geq 0$.

Finally, we show (I4) for any edge $e = uv$ (including loops). Let $\sigma' = w'(e) - \widehat{y'z'}(e)$ and $\sigma = 4n(w(e) - \widehat{yz}(e))$. We claim

$$|\sigma' - \sigma| \leq 4n - 2.$$

The claim gives the desired conclusion (I4). In proof, consider two cases. If e is tight *w.r.t.* y', z', then $\sigma' = 0$. Since σ is a multiple of $4n$ the claim shows it must be 0, so e is tight w.r.t. y, z. If e is not tight, then $|\sigma'| \geq 1$. The claim shows σ cannot have opposite sign from σ'. Thus, either e is dominated w.r.t. both pairs of duals or underrated w.r.t. both.

To prove the claim, let A_u be the minimal blossom containing u and similarly for A_v. Consider two cases.

Case *e is not the base edge of both A_u and A_v*: This case implies there is a blossom B such that $\{A : e \in \gamma(A) \cup I(A)\} = \{A : A \supseteq B, A \text{ a blossom}\}$. In proof, if e is a loop, then clearly we can take B as $A_u = A_v$. If e is a nonloop witness for a blossom B, then B is the desired blossom. If $e = \eta(A_u) \neq \eta(A_v)$, then $B = A_u$.

Using blossom B we have

$$\sigma' = w'(e) - y'(e) - Z'(B) = 4n(w(e) - y(e) - Z(B)) - \mu(e) - \ell(P_u) - \ell(P_v) + r(B) = \sigma \pm r,$$

where $0 \leq r \leq 2n-1$. The claim follows since $2n - 1 \leq 4n - 2$.

Case $e = \eta(A_u) = \eta(A_v)$: Let B be the minimal blossom containing both u and v. Thus, $e = \eta(A)$ for every blossom A with $u \in A \subset B$ or $v \in A \subset B$. Clearly, $\{A : e \in \gamma(A) \cup I(A)\} = \{A : A \text{ a blossom containing } A_u \text{ or } A_v\}$. Thus,

$$\begin{aligned}\sigma' &= w'(e) - y'(e) - Z'(A_u) - Z'(A_v) + Z'(B)\\ &= 4n(w(e) - y(e) - Z(A_u) - Z(A_v) + Z(B)) - 1 - \ell(P_u) - \ell(P_v) + r(A_u) + r(A_v) - r(B)\\ &= \sigma \pm r,\end{aligned}$$

where $0 \leq r \leq 4n - 2$. The claim follows. □

In summary, the algorithm for general costs executes the base algorithm using c' and then transforms the duals. The result is a blossom structure for the matching consisting of all loops. This can be used in the query algorithm of Section 5.4.1 or the representation of Section 5.4.2.

5.4.4 The Time Bound. Regarding efficiency our analysis applies to both the simple base algorithm and the general version. Since $f(V) = 2n$, the complete execution of the f-factor algorithm (on G' and G_r) runs in time $O(n(m + n \log n))$. This dominates the total time to construct the gsp structure. As with the query algorithm, a shortest path for any vertex v can be computed in time proportional to its length, using the data structure of Appendix C.

For a more precise estimate, the set N of negative edges for a conservative cost function is acyclic. Thus $|N| < n$. Let W be the largest magnitude of a given edge cost. Initialize the dual functions by $z \equiv 0$ with no blossoms and

$$y(v) = \begin{cases} W & d(v, N) > 0 \\ 0 & d(v, N) = 0. \end{cases}$$

Match every loop vv where $y(v) = 0$. Now the f-factor algorithm performs $O(|N|)$ searches, using time $O(|N|(m + n \log n))$ assuming $N \neq \emptyset$. This again dominates the time.

THEOREM 5.17. *The generalized shortest-paths structure representing all shortest paths from s is the f-factor algorithm search structure $\mathcal{S}$. It can be constructed in time $O(n(m + n\log n))$. More generally, if $N \neq \emptyset$ is the set of negative cost edges the time is $O(|N|(m + n\log n))$.*

The first time bound is given in Gabow and Sankowski [19]. The second bound shows how the time increases with more negative edges. For example, in a graph with $O(1)$ negative edges the algorithm is as fast as Dijkstra's algorithm, which does not allow negative edges.

A similar dependence on negative edges holds for conservative directed graphs: The single-source shortest-paths problem can be solved in time $O(n_N(m + n\log n))$, for n_N the number of vertices incident to a negative edge. In contrast, the Bellman-Ford algorithm runs in time $O(nm)$ with no dependence on N. To achieve this time bound, we model the digraph G as an undirected bipartite graph: The vertex set is $\{v_1, v_2 : v \in V(G) - s\} + s_2$; the edge set is $\{v_1v_2 : v \in V(G) - s\} \cup \{u_2v_1 : uv \in E(G)\}$, with $c(v_1v_2) = 0, c(u_2v_1) = c(uv)$, $f(v) = 1$ for every vertex. The initialization sets $y(v_i), i = 1, 2$ to W (the largest magnitude of a given cost) if v is on a negative edge else 0. The initial matching consists of the edges v_1v_2 where v is not on a negative edge. A representation of shortest paths from s is constructed similar to the gsp structure; it is exactly the shortest-paths tree.

APPENDIXES

A DUAL ADJUSTMENT STEP FOR EDMONDS' ALGORITHM

To state the dual adjustment step we first review the linear program for perfect matching. Its variables are given by the function $x : E \to \mathbb{R}_+$, which indicates whether or not an edge is matched. The following linear program for maximum matching uses our summing convention, e.g., $x(\delta(v)) = \sum_{e\in\delta(v)} x(e)$:

$$\begin{array}{lllll} \text{maximize} & \sum_{e\in E} w(e)x(e) & & & \\ \text{subject to} & x(\delta(v)) & = & 1 & \text{for every } v \in V \\ & x(\gamma(B)) & \leq & \lfloor \frac{|B|}{2} \rfloor & \text{for every } B \subseteq V \\ & x(e) & \geq & 0 & \text{for every } e \in E. \end{array}$$

The dual LP uses functions $y : V \to \mathbb{R}$, $z : 2^V \to \mathbb{R}_+$. Define $\widehat{yz} : E \to \mathbb{R}$ by

$$\widehat{yz}(e) = y(e) + z\{B : e \subseteq B\}. \tag{A.1}$$

(Note for $e = vw$, $y(e)$ denotes $y(v) + y(w)$ and $z\{B : e \subseteq B\}$ denotes $\sum_{e\subseteq B} z(B)$.)

$$\begin{array}{lllll} \text{minimize} & y(V) + & & & \\ & \sum_{B\subseteq V} \lfloor \frac{|B|}{2} \rfloor z(B) & & & \\ \text{subject to} & \widehat{yz}(e) & \geq & w(e) & \text{for every } e \in E \\ & z(B) & \geq & 0 & \text{for every } B \subseteq V. \end{array}$$

e is *tight* when equality holds in its constraint, $\widehat{yz}(e) = w(e)$. The algorithm maintains the complementary slackness conditions:

$x(e) > 0 \implies e$ is tight.

$z(B) > 0 \implies x(\gamma(B)) = \lfloor \frac{|B|}{2} \rfloor$.

In addition, every edge in a blossom subgraph is tight (so blossoms can be rematched). It is easy to see the dual adjustment step of Figure 18 maintains these conditions.

$\delta_1 \leftarrow \min\{y(e) - w(e) : e = uv \text{ with } u \text{ outer}, v \notin \mathcal{S}\}$
$\delta_2 = \min\{(y(e) - w(e))/2 : e = uv \text{ with } u, v \text{ in distinct outer blossoms}\}$
$\delta_3 = \min\{z(B)/2 : B \text{ an inner blossom of } \overline{\mathcal{S}}\}$
$\delta = \min\{\delta_1, \delta_2, \delta_3\}$
for *every vertex* $v \in \mathcal{S}$ **do**
  if v *is inner* **then** $y(v) \leftarrow y(v) + \delta$ **else** $y(v) \leftarrow y(v) - \delta$
for *every blossom* B *in* $\overline{\mathcal{S}}$ **do**
  if B *is inner* **then** $z(B) \leftarrow z(B) - 2\delta$ **else** $z(B) \leftarrow z(B) + 2\delta$

Fig. 18. Dual adjustment step in Edmonds' algorithm.

B DETAILS FOR b-MATCHING AND f-FACTOR ALGORITHMS

The primal and dual LPs for b-matching are simple generalizations of ordinary matching:

$$\begin{array}{lllll}
\text{maximize} & \sum_{e\in E} w(e)x(e) & & & \\
\text{subject to} & x(\delta(v)) + 2x(\gamma(v)) & = & b(v) & \text{for every } v \in V \\
& x(\gamma(B)) & \leq & \lfloor \frac{b(B)}{2} \rfloor & \text{for every } B \subseteq V \\
& x(e) & \geq & 0 & \text{for every } e \in E,
\end{array}$$

$$\begin{array}{lllll}
\text{minimize} & \sum_{v\in V} b(v)y(v) + & & & \\
& \sum_{B\subseteq V} \lfloor \frac{b(B)}{2} \rfloor z(B) & & & \\
\text{subject to} & \widehat{yz}(e) & \geq & w(e) & \text{for every } e \in E \\
& z(B) & \geq & 0 & \text{for every } B \subseteq V.
\end{array}$$

Similarly, the complementary slackness conditions are essentially unchanged:
$x(e) > 0 \Longrightarrow e$ is tight.
$z(B) > 0 \Longrightarrow x(\gamma(B)) = \lfloor \frac{|b(B)|}{2} \rfloor$.

To refine the complementary slackness condition for $z(B)$, in the primal LP add the constraint for every vertex $v \in B$ to get

$$2x(\gamma(B)) \leq x\{\delta(v) : v \in B\} + 2x\{\gamma(v) : v \in B\} = b(B).$$

If $b(B)$ is even, this gives $x(\gamma(B)) \leq b(B)/2 = \lfloor \frac{|b(B)|}{2} \rfloor$. In other words, the primal constraint for blossom B is redundant. Dropping it means there is no z variable for blossom B. So we can assume $b(B)$ is odd in the complementary slackness condition for z. Thus, complementary slackness can be rewritten as

$z(B) > 0 \Longrightarrow 2x(\gamma(B)) = b(B) - 1$,

i.e., Equation (4.2) holds.

The dual adjustment step of Figure 19 is similar to ordinary matching, extended to the more general definition of eligiblity (e.g., a loop can trigger a blossom step).

Like ordinary matching, the numerical quantities in our algorithm are always half-integers. More precisely assume all given weights $w(e)$ are integral. Assume either every initial y-value is integral or every initial y-value is integral plus 1/2; furthermore, every initial z-value is integral. This assumption holds for common initializations, e.g., $y \equiv \max_{e\in E} w(e)/2$ and $z \equiv 0$. It also holds for the initialization in our strongly polynomial algorithm, Section 4.4. (Note the y-values for BG, i.e., the transportation problem, are integral-valued. So, Equation (4.4) gives half-integral y-values. Doubling the given weight function and these y-values preserves optimality by Equation (4.5). It

$\delta_1 \leftarrow \min\{y(e) - w(e) : e = uv \text{ eligible for } B_u \text{ with } v \notin \mathcal{S}\}$
$\delta_2 = \min\{(y(e) - w(e))/2 : e = uv \text{ eligible for both } B_u \text{ and } B_v\}$
$\delta_3 = \min\{z(B)/2 : B \text{ an inner blossom of } \overline{\mathcal{S}}\}$
$\delta = \min\{\delta_1, \delta_2, \delta_3\}$
for *every vertex* $v \in \mathcal{S}$ **do**
  if B_v *is inner* **then** $y(v) \leftarrow y(v) + \delta$ **else** $y(v) \leftarrow y(v) - \delta$
for *every blossom* B *in* $\overline{\mathcal{S}}$ **do**
  if B *is inner* **then** $z(B) \leftarrow z(B) - 2\delta$ **else** $z(B) \leftarrow z(B) + 2\delta$

Fig. 19. Dual adjustment step for b-matching.

allows using the above initialization.) We will show that throughout the algorithm

$$(\forall v^{\in V})(y(v) \in \mathbb{Z}/2) \text{ and } (\forall B^{\subseteq V})(z(B) \in \mathbb{Z}). \tag{B.1}$$

To prove Equation (B.1), assume it holds before a dual adjustment. Examining the changes of Figure 19 shows it suffices to prove δ is a half-integer. Clearly δ_1 and δ_3 are half-integers. We will show any edge joining two vertices of $\mathcal{S}$ has integral y-value. This makes δ_2 half-integral and completes the proof.

Any tight edge has $\widehat{yz}(e) = w(e)$. So Equation (B.1) (specifically, the integrality of z) implies $y(e) \in \mathbb{Z}$. Any vertex v in $\mathcal{S}$ is joined to a free vertex x by a path P of tight edges. Thus $y(v) + 2y\{u : u \in P - v - x\} + y(x) \in \mathbb{Z}$, i.e., $y(v) + y(x) \in \mathbb{Z}$. Taking any other vertex v' of $\mathcal{S}$ with similar relation $y(v') + y(x') \in \mathbb{Z}$ gives $y(v) + y(v') + y(x) + y(x') \in \mathbb{Z}$. A free vertex is always outer, so its y-value always decreases by δ. So, the initialization implies $y(x) + y(x') \in \mathbb{Z}$. Thus, $y(v) + y(v') \in \mathbb{Z}$ as desired.

The magnitude of numbers computed by the algorithm can be bounded as follows. Let W be the largest magnitude of an edge weight. There is no harm in assuming every initial y and z value has magnitude $\le W$. Let Δ be the total of all dual adjustment quantities δ over the entire algorithm. We claim $\Delta \le Wb(V)$. Clearly, this implies every y and z has magnitude $\le (2b(V) + 1)W$.

To prove the claim, consider any point in the algorithm. The current matching M weighs

$$w(M) = \sum_{e \in M} \widehat{yz}(e) = \sum_{v \in V} d(v, M)y(v) + \sum_{B \in \mathcal{B}} \left\lfloor \frac{b(B)}{2} \right\rfloor z(B). \tag{B.2}$$

In proof, (I4) shows every matched edge is tight. (I5) shows every blossom B with positive z is mature, so $b(B)$ is odd and $|\gamma(V(B), M)| = \frac{b(B)-1}{2} = \lfloor \frac{b(B)}{2} \rfloor$.

Let $b'(v)$ be the remaining degree requirement at v, i.e., $b'(v) = b(v) - d(v, M)$. Equation (B.2) shows the current value of the dual objective function is

$$\sum_{v \in V} b'(v)y(v) + w(M). \tag{B.3}$$

Thus, each dual adjustment decreases the dual objective value by $b'(V)\delta \ge 2\delta$. No other step of the algorithm changes the dual objective (as seen from the LP's formulation of the objective function). Thus, over the entire algorithm, the dual objective decreases by $\ge 2\Delta$.

The initial dual objective is $\le b(V)W$. (This clearly holds in the common case that initially $z \equiv 0$. More generally, Equation (B.3) shows the initially matched edges contribute $|M|W$ to the dual objective and the missing edges contribute $\sum_{v \in V} b'(v)y(v) \le W(b(V) - 2|M|)$.) The final objective is the weight of a maximum b-matching, which is $\ge -Wb(V)/2 \ge -Wb(V)$. So, we always have $2\Delta \le 2b(V)W$, as claimed.

As with ordinary matching, other versions of weighted b-matching have LPs that are minor modifications of the current ones. Correspondingly, minor modifications of our algorithm find such matchings. We illustrate with maximum cardinality maximum weight b-matching (defined in Section 4.3). It is convenient to treat the more general problem of finding a b-matching of maximum weight subject to the constraint that it contains exactly k edges.

The primal LP relaxes the vertex-degree constraint to

$$x(\delta(v)) + 2x(\gamma(v)) \le b(v) \qquad \text{for every } v \in V,$$

and adds the cardinality constraint

$$x(E) = k.$$

The dual problem has a variable c for the cardinality constraint, the left-hand side of the dual edge constraint changes from $\widehat{yz}(e)$ to $\widehat{yz}(e) + c$, and the nonnegativity constraint $y(v) \ge 0$ is added. The additional complementary slackness constraint is

$$y(v) > 0 \Longrightarrow x(\delta(v)) + 2x(\gamma(v)) = b(v) \qquad \text{for every } v \in V.$$

To find such a b-matching, we initialize our algorithm using a common value for every $y(v)$ (e.g., half the maximum edge weight). The algorithm halts after the search that increases the matching size to k. For maximum cardinality maximum weight b-matching, this is the first time a search fails. To get an optimal LP solution, let Y be the common final value for $y(v)$, v free, or 0 if no such vertex exists. (Figure 19 implies that throughout the algorithm all free vertices have the same y-value, and this value is the minimum y-value.) Decrease all y values by Y and set $c = 2Y$. This solves the new LP. (In the dual edge constraint the new y-values decrease $\widehat{yz}(e)$ by $2Y$, which is balanced by the new LP term $c = 2Y$.) We conclude that our algorithm is correct. It also proves the LP formulation is correct.

The LPs for f-factors incorporate limits on the number of copies of an edge as well as $I(B)$ sets of blossoms. For the former it is convenient to treat each copy of an edge separately, i.e., it has its own variable $x(e)$.

maximize	$\sum_{e \in E} w(e)x(e)$			
subject to	$x(\delta(v)) + 2x(\gamma(v))$	$=$	$f(v)$	for every $v \in V$
	$x(\gamma(B) \cup I)$	$\le$	$\lfloor \frac{f(B)+\lvert I\rvert}{2} \rfloor$	for every $B \subseteq V$, $I \subseteq \delta(B)$
	$x(e)$	$\le$	1	for every $e \in E$,
	$x(e)$	$\ge$	0	for every $e \in E$.

The dual LP uses functions $y : V \to \mathbb{R}$, $z : 2^V \times 2^E \to \mathbb{R}_+$. Define $\widehat{yz} : E \to \mathbb{R}$ by

$$\widehat{yz}(e) = y(e) + z\{(B, I) : e \in \gamma(B) \cup I\}.$$

minimize	$\sum_{v \in V} f(v)y(v) + u(E) +$ $\sum_{B \subseteq V, I \subseteq \delta(B)} \lfloor \frac{f(B)+\lvert I\rvert}{2} \rfloor z(B, I)$			
subject to	$\widehat{yz}(e) + u(e)$	$\ge$	$w(e)$	for every $e \in E$
	$u(e)$	$\ge$	0	for every $e \in E$
	$z(B, I)$	$\ge$	0	for every $B \subseteq V$, $I \subseteq \delta(B)$.

In our algorithm, every nonzero z value has the form $z(B, I(B))$ for B a mature blossom. So we use the notation $z(B)$ as a shorthand for $z(B, I(B))$.

Say that e is *dominated, tight,* or *underrated* depending on whether $\widehat{yz}(e)$ is $\ge w(e)$, $= w(e)$, or $\le w(e)$, respectively; *strictly dominated* and *strictly underrated* refer to $> w(e)$ and $< w(e)$, respectively. The complementary slackness conditions for optimality can be written with u eliminated as

$x(e) > 0 \Longrightarrow e$ is underrated,
$x(e) < 1 \Longrightarrow e$ is dominated,
$z(B) > 0 \Longrightarrow x(\gamma(B) \cup I(B)) = \lfloor \frac{f(B)+|I(B)|}{2} \rfloor$.
(The first implication restates complementary slackness for $x(e)$, i.e., $x(e) > 0$ gives equality in the first dual constraint. The second implication restates complementary slackness for $u(e)$, i.e., $u(e) > 0$ gives equality in the third primal constraint.)

The numbers computed by the algorithm are analyzed similar to b-matching. To wit, the same argument shows the algorithm always works with half-integers. The same bound holds for the magnitude of numbers. The proof is the same with two additional remarks to account for the new function $u(E)$:

(*i*) Similar to Equation (B.3), the dual objective function can be rewritten as $\sum_{v\in V} f'(v)y(v) + w(M)$ for $f'(v) = f(v) - d(v, M)$. In proof, recall the optimum u function is defined by setting $u(e)$ equal to the slack in e, $w(e) - \widehat{yz}(e)$, for every edge $e \in M$. So, Equation (B.2) has the analog

$$w(M) = \sum_{e\in M} \widehat{yz}(e) + u(e) = \sum_{v\in V} d(v, M)y(v) + \sum_{B\in\mathcal{B}} \left\lfloor \frac{f(B) + |I(B)|}{2} \right\rfloor z(B) + u(E).$$

This formula also uses the fact that the algorithm maintains complementary slackness for blossoms with positive z value, as verified after Equation (5.11). Substituting this expression into the LP's dual objective gives the desired result.

(*ii*) As in b-matching, examining the LP's dual objective shows no other step changes its value. To verify this for an augment step note that it does not change y or z, or any set $I(B)$ (Lemma 5.3). Furthermore, the augment only changes the matching on tight edges, so $u(E)$ does not change.

Similar to b-matching, our algorithm extends to variants of the maximum f-factor problem. We again illustrate with maximum cardinality maximum weight partial f-factors. The LP is modified exactly as in b-matching. Our modified algorithm and the definition of new LP variables is exactly the same. The only difference in the analysis is that the new complementary slackness conditions for edges are

$x(e) > 0 \Longrightarrow \widehat{yz}(e) + c \le w(e)$,
$x(e) < 1 \Longrightarrow \widehat{yz}(e) + c \ge w(e)$.
As before, the quantity $\widehat{yz}(e) + c$ equals the algorithm's value of $\widehat{yz}(e)$, so these conditions are equivalent to the original ones.

C COMPUTING P-PATHS

This section presents a data structure for blossoms that supports efficient computation of the P-paths for ordinary matching and the P_i-trails for b-matching and f-factors. Specifically, the time to compute P or P_i is proportional to its length. We begin with ordinary matching.

Blossoms form a laminar family with a natural tree representation, as follows. Every maximal blossom B^* has a corresponding tree $T(B^*)$. The root corresponds to B^* and the leaves correspond to $V(B^*)$. The children of any interior node B are the subblossoms B_i of B. $T(B^*)$ is an ordered tree, with order given by the cycles defining blossoms. (Recalling Definition 2.1, the B_i are the contracted vertices on the paths $P(X_i, Y)$. The edges of B are the edges of those paths plus X_0X_1. The corresponding cycle traverses the reverse path $P(Y, X_0)$ followed by edge X_0X_1 and $P(X_1, Y)$.)

$T(B^*)$ has size $O(|V(B^*)|)$. In proof, the leaves of the tree form the set $V(B^*)$. Each interior node is a blossom with three or more subblossoms. So, there are $< |V(B^*)|/2$ interior nodes.

The data structure based on this tree has the following components. Each root node records the base vertex $\beta(B^*)$. The children of any node B form a doubly linked ring. We call this the *sibling*

Procedure R(v, β)
/* adds the edges of $P(v, \beta)$ to global list L
order of edges in L is arbitrary
*/
if $v = \beta$ **then return**
$vw \leftarrow$ the matched edge incident to v
$S_v, S_w \leftarrow$ the siblings joined by vw, with $v \in S_v$, $w \in S_w$
$xy \leftarrow$ the edge from S_w to sibling $\neq S_v$, with $x \in S_w$
add vw and xy to L
R(x, w)
R(y, β)

Fig. 20. Recursive routine to compute P-paths.

ring. Each link records the edge xy of G ($x, y \in V(G)$) that joins the two siblings. In turn, every such edge xy points to the nodes of the two siblings that it joins. Also, every vertex $v \in V(G)$ points to its incident matched edge, if such exists. Finally, the edges of the current matching M are marked as such.

We note some simple properties of the data structure. In $O(1)$ time, the algorithm can go from v to the two siblings joined by contracting v's matched edge. An interior node $B \neq B^*$ with a link corresponding to edge $xy \in M$, $x \in V(B)$ has base vertex x. If neither link from B is matched, then the base vertex of B is that of its first ancestor having such an incident matched edge, or $\beta(B^*)$ if this ancestor does not exist.

The applications of the data structure are based mainly on the $P(v, \beta)$ paths. We now discuss Figure 20, which shows how to compute these paths. Some applications only require the set of edges in $P(v, \beta)$, while others need the edges in path order (details of these applications will be given later). Figure 20 constructs $P(v, \beta)$ as a list of edges L, not necessarily in path order. In Figure 20, L is a global list. Each recursive invocation of R adds its relevant edges to L.

Note how R navigates the blossom tree $T(B^*)$: The first recursive call R(x, w) may descend an arbitrary number of levels in $T(B^*)$. The same holds for the second call, if $y \neq \beta(B)$. If $y = \beta(B)$, the second call may move up an arbitrary number of levels.

The time for one invocation of R is clearly $O(1)$. So, if correct, R finds $P(v, \beta)$ in the desired time bound $O(|P(v, \beta)|)$. We prove correctness by induction on the path length $|P(v, \beta)|$. The base case $v = \beta$ is clearly correct. When $v \neq \beta$, the first edge of $P(v, \beta)$ is vw. Let B denote the blossom containing siblings S_v and S_w. If w is an atom in B, i.e., $S_w = \{w\}$, the first recursive call adds no edges. If S_w is a contracted blossom, the first recursive call is correct by induction. In both cases, the remainder of $P(v, \beta)$ consists of edge xy followed by $P(y, \beta)$. (The matched edge vw shows $x \neq \beta$.) $P(y, \beta)$ is found by the second recursive call, by induction. This completes the correctness proof.

This routine is used in several ways in our implementation of Edmonds' algorithm:

(i) The augment step executes R to find the entire augmenting path. It updates the matched/unmatched status of each edge of L.
(ii) When a grow step adds an inner blossom, say B, R is executed. A bucket sort puts L into path order. $P(x, \beta)$ is added to the supporting forest as described after Definition 3.1.
(iii) Also, in this grow step, the edges of L are marked in their sibling rings. These marks are used in subsequent expand steps for B and its descendants, as follows.

Procedure Ri($\mu_v, v, \mu_\beta, \beta$)
/* assumes P_i has arrived at v on a μ_v-edge
P_i ends entering β on a μ_β-edge
*/
if $v = \beta$ *and* $\mu_v = \mu_\beta$ **then return**
/* now $v = \beta$ implies the remaining path is $P_1(v, v)$ */
$vw \leftarrow$ the μ'_v-edge incident to v
add vw to L
$S_v, S_w \leftarrow$ the siblings joined by vw, with $v \in S_v$, $w \in S_w$
if $w = \beta$ *or* S_w *is atomic or* $w \neq \beta(S_w)$ **then**
    Ri($\mu(vw), w, \mu_\beta, \beta$); **return**
$xy \leftarrow$ the edge from S_w to sibling $\neq S_v$, with $x \in S_w$
/* S_w is a blossom, $w = \beta(S_w) \neq \beta$, $x \neq \beta$ */
add xy to L
1 Ri($\mu(xy), x, \mu'(vw), w$)
Ri($\mu(xy), y, \mu_\beta, \beta$)

Fig. 21. Recursive routine to compute P_i-paths.

Suppose an expand step is done for B. First, consider the processing of $\overline{\mathcal{S}}$ described in Figure 2. The sibling ring for B's children is traversed. The path of subblossoms B_i that replace B in $\overline{\mathcal{S}}$ corresponds to marked links. Subblossoms not among the B_i are no longer in $\mathcal{S}$. Finally, the node for B is discarded so B's children become tree roots.

This procedure can be followed in later expand steps for B_i blossoms that get expanded, and also for their descendants in $T(B_i)$ that get expanded. This works because all corresponding sibling rings have been appropriately marked for $P(x, \beta)$.

Now consider the processing of the supporting tree in an expand step. As described after Definition 3.1, the new outer blossoms B_i, i odd, must be identified. This again is done using the marked links. Additionally, the $T(B)$ leaves descending from such B_i are the new outer vertices that get merged into the supporting tree.

b-matching and f-factors. The data structure is similar to ordinary matching. The biggest difference is that each vertex $v \in V(G)$ has an additional pointer, to an unmatched edge. We call these pointers the *M-pointer* and the *$\overline{M}$-pointer*. To define them let A be the minimal blossom containing v. If $v \neq \beta(A)$ its pointers correspond to the two edges of $\delta(v, C(A))$. If $v = \beta(A)$, one pointer is the first edge of $C(A)$. (The last edge would do just as well.) The other is the edge $\eta(A)$ of Definition 5.1, i.e., the edge of $\delta(v) \cap \delta(V(A))$ of opposite M-type from A. Note these pointers are easily maintained in an augment step (wherein v may change back and forth between base vertex and nonbase).

Other changes to the data structure are minor. A minimal blossom may be a loop vv. Its child is the leaf vertex v. So our bound on the number of interior nodes increases to $3|V(B^*)|/2$. But the size of $T(B^*)$ remains $O(|V(B^*)|)$. In the sibling ring, the node $\alpha(B)$ occurs only once, not twice.

We use the letter μ to refer to an M-type M or $\overline{M}$. μ' refers to the opposite type. An example is that for any edge e, $\mu(e)$ denotes the M-type of e, and $\mu'(e)$ is the opposite M-type.

Figure 21 gives the recursive routine to compute P_i trails. Consider the parameter μ_v of Ri. When an invocation of Ri arrives at v by traversing an edge e ending at v, it issues a call with

μ_v set to $\mu(e)$. In other cases the parameter μ_v is set to the M-type that mimics this arrival. For example, in the initial call to Ri, if P_i begins with a type μ edge then μ_v is set to μ'. It is for this reason that the value i is not used in our algorithm.

The test in Figure 21 to decide if $w = \beta(S_w)$ is implemented in $O(1)$ time by checking if vw is the $\mu(vw)$-pointer of w.

The proof of correctness is similar to R. Observe that the special case $w = x$ is handled correctly in line 1: If $\mu(vw) = \mu(xy)$ then $P_1(w, w)$ is correctly added to L, since line 1 can be rewritten as $\texttt{Ri}(\mu(vw), w, \mu'(vw), w)$. If $\mu(vw) \neq \mu(xy)$, then P_i goes directly from vw to xy. This again corresponds to line 1, which can now be rewritten as $\texttt{Ri}(\mu'(vw), w, \mu'(vw), w)$. Finally, note that this discussion includes the case of a loop blossom vv.

The Ri routine is used the same way as R, with just simple modifications. In an augment step, for edge xy the augmenting trail is traversed using two calls to Ri, both having first argument $\mu_v = \mu(xy)$. The same first argument is used in a grow step for edge xy. In constructing the supporting tree, we eliminate maximal subtrails $P_1(\beta(A), \beta(A))$ (recall the paragraphs preceding Theorem 4.14). This is easily done with a bucket sort.

D GROW/EXPAND STEPS

This section presents a simple data structure to handle grow and expand steps. To illustrate the difficulty first consider ordinary matching. At any point in a search, for any vertex $v \in V$ define $slack(v)$ as the smallest slack in an unmatched edge from an outer node to v. If $v \notin \mathcal{S}$ then $slack(v)$ may decrease because of new outer nodes. Also, once $slack(v) < \infty$, dual adjustments automatically decrease $slack(v)$. When $slack(v)$ becomes zero a grow step can be performed to make B_v inner. But if B_v is a blossom, it may become inner before $slack(v)$ becomes zero. This blossom may later get expanded. This may cause v to leave $\mathcal{S}$. If not some smaller blossom containing v may get expanded causing v to leave $\mathcal{S}$. Continuing in this fashion v may oscillate in and out of $\mathcal{S}$, becoming eligible and ineligible for grow steps. This makes tracking potential grow steps nontrivial. Note there is no such complication for grow steps using a matched edge to add a new outer node, since matched edges are always tight and outer nodes never leave $\mathcal{S}$.

The same overview applies to b-matching. f-factors are more general, since matched edges needn't be tight. We first present the algorithm that applies to ordinary matching and b-matching. Then we extend the algorithm to f-factors.

Data Structures. We represent the laminar structure of blossoms using the trees $T(B^*)$ of the last section, as follows. At the start of a search, the current blossoms (from previous searches) form a tree $\mathcal{B}$. The root of $\mathcal{B}$ corresponds to V. The children of the root are the maximal blossoms and the atoms not in a blossom. Each maximal blossom B^* has $T(B^*)$ from last section as its subtree. In addition, each blossom $B \in \mathcal{B}$ records its base edge $\eta(B)$.

Recall (Section 3.1) the rank of a $\mathcal{B}$-node B is $r(B) = \lfloor \log |V(B)| \rfloor$. A $\mathcal{B}$-child of B is *small* if it has rank $< r(B)$, else *big*. Clearly, B has at most one big child. So, the rank $r(B)$ descendants of B form a path P that starts at B. Each node on P except B is the big child of its parent.[12] The data structure marks each node as big or small.

Furthermore, a small child of any node on the path P is called a *small component* of B. If B is a blossom, then

$$V(B) = \cup\{V(A) : A \text{ a small component of } B\}.$$

(This fails if B is a leaf of $\mathcal{B}$. Such a B has no children or components.)

[12] P is a slight variant of the "heavy path" of References [24, 35].

As before, $\overline{G} = (\overline{V}, \overline{E})$ denotes the current contracted graph in the algorithm. $\overline{V}$ consists of currently maximal blossoms, and atoms not belonging to a current blossom. As usual B_v denotes the $\overline{V}$-vertex currently containing v. The main task for the data structure is tracking $slack(v)$ values, and this requires tracking B_v. The values $node(v)$, to be defined, will identify B_v in $O(1)$ time. $node(v)$ values are also used in blossom and augment steps to compute paths in $\overline{\mathcal{S}}$.

Recall these definitions from the data structure for numerical quantities (given in the last subsection of Section 4.3): Δ is the sum of all dual adjustment quantities δ in the current search. Any outer vertex v has a quantity $Y(v)$ such that the current value of $y(v)$ is $Y(v) - \Delta$. A global Fibonacci heap $\mathcal{F}$ has entries for candidate grow, blossom, and expand steps. An entry's key is the value of Δ when the corresponding step can be executed.

To compute current y and z values for nonouter vertices, we use an auxiliary quantity $DEL(B)$. It tracks z-values of expanded blossoms, that have been converted into y-values. To define this quantity, let y_0 and z_0 denote the dual functions at the start of the current search. The algorithm records the quantity

$$Y(v) = y_0(v)$$

for every $v \in V$. Every node B of $\mathcal{B}$ is labeled with the quantity

$$DEL(B) = \frac{1}{2} z_0\{A : A \text{ a proper ancestor of } B \text{ in } \mathcal{B}\}. \tag{D.1}$$

Observe that when B is a $\overline{V}$-vertex, $DEL(B)$ is the total of all dual adjustments made while B was properly contained in an inner blossom. At any point in time current y values are given by

$$y(v) = \begin{cases} Y(v) + DEL(B_v) & B_v \text{ not in } \overline{\mathcal{S}} \\ Y(v) + DEL(B_v) + \Delta - \Delta_0(B_v) & B_v \text{ an inner node,} \end{cases} \tag{D.2}$$

where $\Delta_0(B)$ denotes the value of Δ when $\overline{V}$-vertex B became an inner node. We will compute $y(v)$ in $O(1)$ time when it is needed. To do this, we must identify B_v in $O(1)$ time. This is done using the pointer $node(v)$ to be described.

We track the best candidate edges for grow steps from outer nodes using a system of Fibonacci heaps. At any point in the algorithm every nonouter $\overline{V}$-blossom B has a Fibonacci heap $\mathcal{F}_B$. The nodes of $\mathcal{F}_B$ are the small components of B. Thus if B is not a node of $\overline{\mathcal{S}}$, the smallest slack of an unmatched edge for a grow step to B is the smallest value $slack(v)$, $v \in V(A)$, A a blossom or atom with a node in $\mathcal{F}_B$.

The data structure must also handle nonouter $\overline{V}$-atoms B. For uniformity we assume atoms are handled like blossoms—they have a Fibonacci heap of one node, the atom itself. We will not dwell on this case; the reader can make the obvious adjustments for nonouter $\overline{V}$-atoms.

Returning to the general case, the data structure does not explicitly store values $slack(v)$ since they change with every dual adjustment. Instead it stores offsetted versions of related quantities as follows.

Whenever B_v is not in $\overline{\mathcal{S}}$, the slack in an unmatched edge uv with B_u outer is

$$y(u) + y(v) - w(uv) = (Y(u) - \Delta) + (Y(v) + DEL(B_v)) - w(uv).$$

(This does not depend on prior history, i.e., when u first became outer or v's movements in and out of $\mathcal{S}$.) So, the data structure stores the quantity

$$SLACK(v) = \min\{Y(u) + Y(v) - w(uv) : B_u \text{ outer}, uv \in E - M\},$$

for every vertex v where B_v is not outer. The expression for a given edge uv never changes in value even as B_u changes. The data structure also records the minimizing edge uv. $SLACK(v)$ and

its minimizing edge are updated as new outer nodes are created. At any point in time when v is not in $\mathcal{S}$, the current value of $slack(v)$ is

$$slack(v) = SLACK(v) - \Delta + DEL(B_v). \tag{D.3}$$

The key of a node A in $\mathcal{F}_B$ is

$$key(A, \mathcal{F}_B) = \min\{SLACK(v) : v \in V(A)\}. \tag{D.4}$$

At any point in time when B is not in $\overline{\mathcal{S}}$, the current smallest slack of an unmatched grow step edge to B is $find\_min(\mathcal{F}_B) - \Delta + DEL(B)$. Thus, a grow step for B can be done when $\Delta = find\_min(\mathcal{F}_B) + DEL(B)$. So, every $\overline{V}$-vertex B that is not a node of $\overline{\mathcal{S}}$ has an entry in the global heap $\mathcal{F}$, with key maintained as $find\_min(\mathcal{F}_B) + DEL(B)$.

For every vertex $v \in V$, $node(v)$ is the unique ancestor of v that is currently a node of some heap $\mathcal{F}_B$. $node(v)$ is used in Equation (D.4) to maintain keys in $\mathcal{F}_B$ (i.e., $node(v)$ gives A in Equation (D.4)). $node(v)$ is also used in Equation (D.2) to determine the current blossom B_v. Specifically, $node(v)$ is in the heap $\mathcal{F}_{B_v}$.

Algorithms. When a new outer node B is created, every unmatched edge uv, $u \in B$ is examined. $SLACK(v)$ is decreased if appropriate. This may trigger a $decrease\_key$ for $node(v)$ in $\mathcal{F}_{B_v}$. This may in turn trigger a $decrease\_key$ for B_v in $\mathcal{F}$, if B_v is currently not in $\overline{\mathcal{S}}$.

When a grow step adds a blossom B to $\overline{\mathcal{S}}$, the node for B in $\mathcal{F}$ is deleted. Whether B becomes inner or outer, it never gets reinserted in $\mathcal{F}$ in this search. If B becomes inner the value $\Delta_0(B)$ is recorded. If B becomes outer, the values $y(v), v \in V(B)$ are required to redefine $Y(v)$ (recall from Section 4.3). This is done using the first alternative of Equation (D.2). If B becomes inner and later becomes outer in a blossom step, $Y(v)$ is redefined using the second alternative of Equation (D.2).

Consider an expand step for an inner blossom B. The $\mathcal{B}$-children of B (i.e., the nodes of $C(B)$) become $\overline{V}$-vertices, and we must update the data structure for them. Let B' be the big $\mathcal{B}$-child of B, if it exists. For every $\mathcal{B}$-child $A \neq B'$ of B, delete the node A of $\mathcal{F}_B$. Initialize a new F-heap $\mathcal{F}_A$ as follows (modifying appropriately if A is atomic):

> For each small component D of A, create a node in $\mathcal{F}_A$. For every $v \in V(D)$ update $node(v)$ to D. Assign $key(D, \mathcal{F}_A) \leftarrow \min\{SLACK(v) : v \in V(D)\}$.

Let the new heap $\mathcal{F}_{B'}$ be the (updated) heap $\mathcal{F}_B$. Insert the $\mathcal{B}$-children of B that are no longer nodes of $\overline{\mathcal{S}}$ as entries in $\mathcal{F}$. For the $\mathcal{B}$-children that are inner nodes of $\overline{\mathcal{S}}$ record their Δ_0 value. Process $\mathcal{B}$-children that are outer nodes of $\overline{\mathcal{S}}$ as above.

The main observation for correctness of the expand procedure is that $\mathcal{F}_{B'}$ is the desired heap for B'. This follows since the small components of B' are those of B minus the small children of B.

It is easy to see the total time used in the course of an entire search is $O(m + n \log n)$. When a small child A becomes maximal it is charged $O(\log n)$ to account for its deletion from $\mathcal{F}_B$. For D a small component of A, each vertex $v \in V(A)$ is charged $O(1)$ for resetting $node(v)$ and examining $SLACK(v)$. (The new $node(v)$ values are found by traversing the subtree of A in the blossom tree $\mathcal{B}$. The traversal uses time proportional to the number of leaves, i.e., $O(1)$ time for each vertex v.) v moves to a new small component D $O(\log n)$ times, so this charge totals $O(n \log n)$ for all vertices. Finally, and most importantly, $decrease\_key$ uses $O(1)$ amortized time in a Fibonnaci tree.

f-factors. Two new aspects of f-factors are that matched edges needn't be tight and edges can be in I-sets. We will use some simple facts about I-sets.

Lemma D.1. *Consider blossoms A, B with $V(A) \subseteq V(B)$, and edge $e \in \delta(A) \cap \delta(B)$.*
(i) $e = \eta(A) \iff e = \eta(B)$.
(ii) $e \in I(A) \iff e \in I(B)$.

Proof. (*i*) Consider three cases for A.

Case $A \not\subseteq \alpha(B)$: This makes $\eta(A) \in \gamma(B)$. So, $e \in \delta(B)$ implies $e \neq \eta(A)$. Also, $e \in \delta(A)$ implies $e \neq \eta(B)$.

Case $A = \alpha(B)$: This makes $\eta(A) = \eta(B)$. Hence $e = \eta(A)$ iff $e = \eta(B)$.

Case $A \subset \alpha(B)$: Edge e of the hypothesis is in $\delta(A) \cap \delta(\alpha(B))$. By induction $e = \eta(A) \iff e = \eta(\alpha(B))$. Since $\eta(\alpha(B)) = \eta(B)$ this implies (*i*).
(*ii*) By (*i*) there are two possibilities:

Case $e \neq \eta(A), \eta(B)$: $e \in I(A) \iff e \in M \iff e \in I(B)$.

Case $e = \eta(A) = \eta(B)$: $e \in I(A) \iff e \notin M \iff e \in I(B)$. □

Now, observe an edge $e = uv \in I(B_v)$ has

$$z_0\{A : V(A) \subseteq V(B_v),\ e \in I(A)\} = z_0\{A : v \in V(A) \subseteq V(B_v)\} = 2(DEL(v) - DEL(B_v)). \tag{D.5}$$

The second equation is trivial and the first follows immediately part (*ii*) of the lemma.

The analog of the previous definition of *slack* is

$$slack(v) = \min\{|\widehat{yz}(uv) - w(uv)| : uv \in E \text{ eligible at } u\}. \tag{D.6}$$

(Recall Lemma 5.9 and its terminology.) As in Lemma 5.8, define a sign σ as -1 if $uv \in M$ else $+1$, so any edge uv has $|\widehat{yz}(uv) - w(uv)| = \sigma(\widehat{yz}(uv) - w(uv))$.

The highest level outline of the data structure is as before: We track *slack* by maintaining the invariant Equation (D.3), where the stored quantity $SLACK(v)$ will be defined below. We define keys in $\mathcal{F}_B$ and $\mathcal{F}$ exactly as before, e.g., Equation (D.4). The invariant implies that for any blossom B not in $\overline{\mathcal{S}}$, the current smallest *slack* of a grow step edge to B is $find\_min(\mathcal{F}_B) - \Delta + DEL(B)$. So, the data structure gives the correct value for the next dual adjustment.

Our definition of $SLACK(v)$ involves two quantities $IU(uv)$ and $IV(uv)$ that account for the contributions of I-edges to the slack of uv, IU at the u end and IV at the v end. We will define IU and IV to be fixed, stored quantities so the following two relations hold. At any time when $v \notin \mathcal{S}$ and B_v is the $\overline{V}$-vertex currently containing v,

$$y(v) + z\{A : v \in V(A),\ uv \in I(A)\} = Y(v) + IV(uv) + \sigma DEL(B_v). \tag{D.7}$$

At any time after uv becomes eligible at u,

$$y(u) + z\{A : u \in V(A),\ uv \in I(A)\} = Y(u) + IU(uv) - \sigma\Delta. \tag{D.8}$$

IU and IV are computed at the instant uv becomes eligible at u. Thereafter they are not changed. The only terms on the right-hand side of Equations (D.7)–(D.8) that change with time are $DEL(B_v)$ and Δ.

Define

$$SLACK(v) = \min\{\sigma(Y(u) + Y(v) + IU(uv) + IV(uv) - w(uv)) : uv \in E \text{ eligible at } u\}.$$

Let us show the above relations imply the desired invariant Equation (D.3) for *SLACK*. Adding the two equations and multiplying by σ implies that at any point in time when uv is eligible and $v \notin S$,

$$|\widehat{yz}(uv) - w(uv)| = \sigma(Y(u) + IU(uv) + Y(v) + IV(uv) - w(uv)) - \Delta + DEL(B_v).$$

Applying this for every edge uv in the definition of *SLACK* gives Equation (D.3) as desired.

It remains to specify IV and IU. The contribution at the nonouter end v is defined by

$$IV(uv) = \begin{cases} 0 & uv \notin M \cup \eta(B_v) \\ 2DEL(v) & uv \in M - \eta(B_v) \\ 2DEL(B_v) & uv = \eta(B_v) \in M \\ 2(DEL(v) - DEL(B_v)) & uv = \eta(B_v) \notin M. \end{cases}$$

We will discuss this definition using the following terminology. Recall that the algorithm computes $IV(uv)$ when uv becomes eligible at u. $IV(uv)$ is defined using the $\overline{V}$-vertex B_v at that time. However, we must verify Equation (D.7) whenever $v \notin \mathcal{S}$, so B_v may change. To distinguish the two possibilities say the *defining* B_v is used to compute $IV(uv)$, and a *useful* B_v is one that may be required later on in Equation (D.7) to establish the invariant Equation (D.3). The defining B_v is useful iff $v \notin \mathcal{S}$ when uv becomes eligible at u. Clearly, a useful B_v is a subset of the defining B_v.

To prove the definition is correct, we will analyze the four cases separately. We will show that if the defining B_v belongs to that case, so does every useful B_v. Then we will show Equation (D.7) is satisfied for every useful B_v. To do this we will compute the value of the left-hand side of Equation (D.7) and deduce the correct value of $IV(uv)$ by comparing to the right-hand side.

We start with two relations that hold for every case. Just as before, whenever $v \notin \mathcal{S}$ the current value of $y(v)$ is

$$Y(v) + DEL(B_v),$$

since every dual adjustment increases $y(v)$ by δ. Also, Equation (D.5) shows that when $uv \in I(B_v)$ the z contribution to the left-hand side of Equation (D.7) is

$$z_0\{A : v \in V(A) \subseteq V(B_v)\} = 2(DEL(v) - DEL(B_v)).$$

Case $uv \notin M \cup \eta(B_v)$: We are assuming this case holds for the defining B_v. But any useful B_v, say B, also has uv unmatched and $uv \neq \eta(B)$ (by Lemma D.1(i)). So this case holds for every useful B.

Now we establish Equation (D.7) for any useful B_v. The contribution to the left-hand side of Equation (D.7) is $y(v) = Y(v) + \sigma DEL(B_v)$. This follows since $\sigma = 1$ (because $uv \notin M$) and this case implies $uv \notin I(B_v)$ (so there is no z contribution). Comparing to the right-hand side of Equation (D.7) shows $IV(uv) = 0$ as claimed.

Case $uv \in M - \eta(B_v)$: Again we are assuming this holds for the defining B_v. Just as before, this case holds for every useful B_v.

Consider any useful B_v. If B_v is a blossom, then $uv \in I(B_v)$. So the z contribution is $2(DEL(v) - DEL(B_v))$. This also holds if B_v is atomic since the z contribution is zero. Since $uv \in M$, $\sigma = -1$. Adding the y and z contributions to the left-hand side of Equation (D.7) gives total contribution

$$(Y(v) + DEL(B_v)) + 2(DEL(v) - DEL(B_v)) = Y(v) + 2DEL(v) + \sigma DEL(B_v).$$

Thus, $IV(uv) = 2DEL(v)$, again independent of B_v.

The last two cases have $uv = \eta(B_v)$ for the defining B_v. If $v \in \mathcal{S}$ when $IV(uv)$ is defined, then *wlog* B_v is inner. Since $v = \beta(B_v)$, v will remain in $\mathcal{S}$ for the rest of the search. So uv is irrelevant to the data structure. If $v \notin \mathcal{S}$ then B_v is itself the first useful B_v. The first time this B_v becomes a node of $\overline{\mathcal{S}}$, if it becomes inner the preceding argument shows there are no other useful B_v's. The same holds if B_v becomes outer. In summary, we have shown that in the last two cases, every useful B_v belongs to the same case.

Case $uv = \eta(B_v) \in M$: Since $uv \notin I(B_v)$ there is no z contribution (by Lemma D.1(ii)). So, the total contribution is $y(v) = Y(v) + DEL(B_v) = Y(v) + 2DEL(B_v) + \sigma DEL(B_v)$. Thus $IV(uv) = 2DEL(B_v)$.

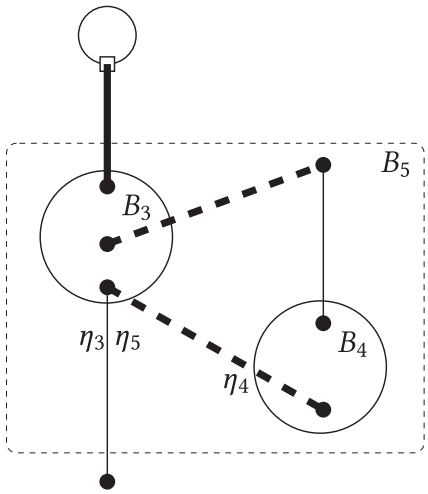

Fig. 22. Precursor to structure of Figure 12.

Case $uv = \eta(B_v) \notin M$: This makes $uv \in I(B_v)$ so there is a z contribution. The total contribution is

$$(Y(v) + DEL(B_v)) + 2(DEL(v) - DEL(B_v)) = Y(v) + 2(DEL(v) - DEL(B_v)) + \sigma DEL(B_v).$$

Thus, $IV(uv) = 2(DEL(v) - DEL(B_v))$.

Remark. It might seem that the cases for $uv = \eta(B_v)$ are subject to a simplification because this edge is often tight. Specifically, if B_v was not a maximal blossom at the beginning of the current search then $\eta(B_v)$ is tight when the search starts. $\eta(B_v)$ will still be tight when B_v becomes maximal. But this needn't be the case when $\eta(B_v)$ becomes eligible at u. For instance, suppose a search starts out with the structure of Figure 22. Then the inner blossom B_5 gets expanded to give part of Figure 12, where $\alpha_2 = \eta_4 = \eta(B_4)$. As mentioned (Section 5.2, Examples, Strictly Underrated Edges) a dual adjustment makes α_2 strictly underrated. A subsequent expansion of B_3 may make α_2 eligible, but still strictly underrated.

Turning to the contribution at the $\overline{S}$ end u, define

$$IU(uv) = \begin{cases} DEL(B_u) - \Delta_0(B_u) & B_u \text{ inner}, uv \in M \\ 2DEL(u) - DEL(B_u) + \Delta_0(B_u) & B_u \text{ inner}, uv = \eta(B_u) \notin M \\ 0 & B_u \text{ outer}, uv \notin M \\ -2\Delta_0(B_u) & B_u \text{ outer}, uv \in M, A_u \text{ atomic} \\ 2(DEL(u) - DEL(A_u) - 2\Delta_0(B_u) + \Delta_0(A_u)) & B_u \text{ outer}, uv \in M, A_u \text{ a blossom.} \end{cases}$$

A_u is defined below.

To verify correctness, let Δ_0 be the value of Δ when uv first becomes eligible at u. ($\Delta_0 = \Delta_0(B_u)$ in the preceding definition.) We will show Equation (D.8) holds at that instant. Thereafter, uv remains eligible (Lemma 5.9) so Equation (5.5) shows the left-hand side of Equation (D.8) changes by $-\sigma\delta$ in every dual adjustment. This matches the change in the right-hand side. Thus, Equation (D.8) continues to hold in every dual adjustment.

Case B_u *inner*, $uv \in M$: This makes $uv \notin I(B_u)$. (There are two cases: If B_u is a blossom, then $uv = \eta(B_u)$ since uv is eligible, and $\eta(B_u) \notin I(B_u)$. If B_u is atomic, then $I(B_u) = \emptyset$.) Thus, the con-

tribution is

$$y(u) = Y(u) + DEL(B_u) = Y(u) + DEL(B_u) - \Delta_0(B_u) - \sigma\Delta_0.$$

Thus, $IU(uv) = DEL(B_u) - \Delta_0(B_u)$.

Case B_u *inner*, $uv = \eta(B_u) \notin M$: This makes B_u a blossom and $uv \in I(B_u)$. The contribution for $y(u)$ is the same as the previous case. The contribution for z is $2(DEL(u) - DEL(B_u))$. The total contribution is $(Y(u) + DEL(B_u)) + 2(DEL(u) - DEL(B_u)) = Y(u) + 2DEL(u) - DEL(B_u) + \Delta_0(B_u) - \sigma\Delta_0$. Thus, $IU(uv) = 2DEL(u) - DEL(B_u) + \Delta_0(B_u)$.

We are left with the possibility that uv first becomes eligible at u when B_u becomes an outer node. If B_u is a blossom, then $uv \neq \eta(B_u)$, since an outer blossom has $\eta(B_u) = \tau(B_u) \in \overline{\mathcal{S}}$.

Recall that when B_u is formed we redefine $Y(u)$ to be the current value of $y(u)$ plus $\Delta_0(B_u)$. Hence at any time after that

$$y(u) = Y(u) - \Delta.$$

Also, since we are interested in Equation (D.8) when B_u is formed, $z(B_u) = 0$ on the left-hand side.

Case B_u *outer*, $uv \notin M$: There is no z contribution in Equation (D.8). (This is by definition if B_u is atomic. If B_u is a blossom we have noted $uv \neq \eta(B_u)$.) So the total contribution is $y(u) = Y(u) - \Delta_0(B_u) = Y(u) - \sigma\Delta_0$. Thus $IU(uv) = 0$.

Case B_u *outer*, $uv \in M$: B_u is an outer blossom. It is not atomic since uv is matched and eligible for B_u. We define A_u as the $\overline{V}$-vertex containing u right before B_u is formed. There are several possibilities for B_u and A_u:

- B_u *is formed in a grow step*: $A_u = B_u$. (A_u changes from non-$\overline{\mathcal{S}}$ to an outer blossom.)
- B_u *is formed in a blossom step*: A_u is an outer atom or an inner blossom. (A_u is not an inner atom since such vertices already have uv eligible at u and are treated in the first case.)
- B_u *is formed in an expand step*: $A_u = B_u$. (A_u becomes a maximal outer blossom in the expand.)

To start the analysis, first suppose A_u is an outer atom (from a blossom step). An atom has no z contribution. So the left-hand side of Equation (D.8) is $(Y(u) - \Delta_0(B_u)) - \Delta_0(B_u) - \sigma\Delta_0$, and $IU(uv) = -2\Delta_0(B_u)$.

The remaining possibilities all have A_u a blossom. So $uv \in I(A_u)$. The z contribution in Equation (D.8) is

$$z_0\{A : u \in V(A) \subseteq V(A_u)\} = 2(DEL(u) - DEL(A_u)).$$

If A_u becomes outer in a blossom step, A_u is inner until it is absorbed into B_u. In that time interval the z contribution decreases by

$$2(\Delta_0(B_u) - \Delta_0(A_u)).$$

This expression is also valid in the other possibilities – $A_u = B_u$ so the expression vanishes and A_u immediately becomes outer.

Combining the expressions shows the left-hand side of Equation (D.8) is

$$\begin{aligned}&(Y(u) - \Delta_0(B_u)) + 2(DEL(u) - DEL(A_u) - (\Delta_0(B_u) - \Delta_0(A_u)))\\ &= Y(u) + 2(DEL(u) - DEL(A_u) - 2\Delta_0(B_u) + \Delta_0(A_u)) - \sigma\Delta_0.\end{aligned}$$

Thus, $IU(uv) = 2(DEL(u) - DEL(A_u) - 2\Delta_0(B_u) + \Delta_0(A_u))$. This concludes the analysis of IU.

The only changes to the algorithm are obvious ones for examining edges: Matched edges must be examined and added to the data structure. IU and IV quantities must be computed. It is easy to see the latter uses $O(1)$ time per edge. So the timing estimate is not affected.

E SHORTEST-PATHS REPRESENTATION FOR REAL-VALUED COSTS

This appendix proves the shortest-paths representation of Section 5.4.2 exists for arbitrary real-valued conservative edge-cost functions. Specifically, we show how to execute the f-factor searches of the base algorithm on a perturbation of the given cost function, so the final matching consists of all the loops, and the blossom structure and dual functions are valid for the given unperturbed costs. The analysis of the base algorithm in Sections 5.4.1–5.4.2 shows this suffices to construct the gsp-representation.

Conceptually increase each edge cost $c(e)$ by the same unknown positive quantity ϵ. The f-factor algorithm will compute all numeric quantities as expressions of the form $r + s\epsilon$, where r and s are known real-valued quantities and ϵ is a symbol. This is easily done since numeric values are only manipulated using simple arithmetic operations in the dual adjustment step. Call the resulting f-factor search the *symbolic algorithm*.

In contrast, call a version of the f-factor search that uses actual numeric values a *numeric algorithm*. We will maintain a value $\epsilon_0 > 0$ and call every value $\epsilon \in [0, \epsilon_0)$ *relevant*. Every relevant value has a corresponding numeric algorithm, wherein each numeric quantity $r + s\epsilon$ has its actual numeric value. We will maintain the invariant that every relevant numeric execution performs the same sequence of grow, blossom, expand, and dual adjustment steps as the symbolic algorithm. Let us describe how this is done.

The symbolic algorithm compares quantities using lexicographic order $\prec$, i.e., $r + s\epsilon \prec r' + s'\epsilon$ iff $r < r'$ or $r = r$ and $s < s'$. For any quantity $r + s\epsilon$ define

$$\rho = \begin{cases} (r' - r)/(s - s') & r < r' \text{ and } s > s' \\ \infty & \text{otherwise.} \end{cases}$$

We use this simple fact:

Proposition E.1. *$r + s\epsilon \prec r' + s'\epsilon$ implies that any $\epsilon \in (0, \rho)$ has $r + s\epsilon < r' + s'\epsilon$, and weak inequality holds when $\epsilon = 0$.*

Proof. Weak inequality for $\epsilon = 0$ follows from the assumed relation $r \le r'$. For $\epsilon > 0$ examine the possibilities $r = r'$ and $s < s'$, $r < r'$ and $s \le s'$, and $r < r'$ and $s > s'$. □

Consider the dual adjustment step. Each slack value, e.g., $|\widehat{yz}(e) - w(e)|/2$ in the set defining δ_2, has the symbolic form $r + s\epsilon$. The symbolic algorithm compares slack values to find the minimum. Each comparison with resulting outcome $r + s\epsilon \prec r' + s'\epsilon$ adjusts the value of ϵ_0 to $\min\{\epsilon_0, \rho\}$. The result is that the smallest slack identified by the symbolic algorithm, say $\delta = \overline{r} + \overline{s}\epsilon$, can be chosen as the smallest slack by every relevant numeric algorithm, by the preceding proposition. So all relevant numeric executions set $\delta = \overline{r} + \overline{s}\epsilon$. All algorithms, symbolic and relevant, then proceed in the same way: Eligible edges have their slacks decrease by $\overline{r} + \overline{s}\epsilon$. Then the same grow, blossom, and expand steps are executed. (We assume any ties are broken the same way by all algorithms.)

Applying this reasoning to every dual adjustment step, we see that all relevant executions construct the same final f-factor. It consists of all the loops, since $\epsilon > 0$ implies no cycle of given edges has cost 0. Thus, the execution for $\epsilon = 0$ halts with the matching consisting of all loops, as desired.

ACKNOWLEDGMENTS

The author thanks Bob Tarjan for some fruitful early conversations, as well as Jim Driscoll. Also, thanks to an anonymous referee for a careful reading and many suggestions.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Saddle River, NJ.

[3] R. P. Anstee. 1987. A polynomial algorithm for b-matchings: An alternative approach. *Inform. Process. Lett.* 24 (1987), 153–157.

[4] M. O. Ball and U. Derigs. 1983. An analysis of alternative strategies for implementing matching algorithms. *Networks* 13, 4 (1983), 517–549.

[5] R. Cole and R. Hariharan. 2005. Dynamic LCA queries on trees. *SIAM J. Comput.* 34, 4 (2005), 894–923.

[6] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. 1998. *Combinatorial Optimization*. Wiley and Sons, New York.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2001. *Introduction to Algorithms* (2nd ed.). McGraw–Hill, New York.

[8] W. H. Cunningham and A. B. Marsh. 1978. A primal algorithm for optimum matching. *Math. Programming Study* 8 (1978), 50–72.

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

[10] M. L. Fredman and R. E. Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34, 3 (1987), 596–615.

[11] H. N. Gabow. 1973. *Implementations of Algorithms for Maximum Matching on Nonbipartite Graphs*. Ph.D. Dissertation. Comp. Sci. Dept., Stanford Univ., Stanford, CA.

[12] H. N. Gabow. 1976. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *J. ACM* 23, 2 (1976), 221–234.

[13] H. N. Gabow. 1983. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computation*. ACM Press, New York, NY, 448–456.

[14] H. N. Gabow. 1985. A scaling algorithm for weighted matching on general graphs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 90–100.

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

[16] H. N. Gabow. 2017. A data structure for nearest common ancestors with linking. *ACM Trans. Algorithms* 13, 4 (2017), 28 pages. Article 45.

[17] H. N. Gabow, Z. Galil, and T. H. Spencer. 1989. Efficient implementation of graph algorithms using contraction. *J. ACM* 36, 3 (1989), 540–572.

[18] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* 6, 2 (1986), 109–122.

[19] H. N. Gabow and P. Sankowski. 2013. Algebraic algorithms for b-matching, shortest undirected paths, and f-factors. In *Proceedings of the 54th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Long Beach, CA, 137–146. Revised version, 2016: Algorithms for weighted matching generalizations I: Bipartite graphs, b-matching, and unweighted f-factors; Algorithms for weighted matching generalizations II: f-factors and the special case of shortest paths.

[20] H. N. Gabow and R. E. Tarjan. 1985. A linear–time algorithm for a special case of disjoint set union. *J. Comp. and System Sci.* 30, 2 (1985), 209–221.

[21] H. N. Gabow and R. E. Tarjan. 1991. Faster scaling algorithms for general graph matching problems. *J. ACM* 38, 4 (1991), 815–853.

[22] Z. Galil, S. Micali, and H. N. Gabow. 1986. An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM J. Comput.* 15, 1 (1986), 120–130.

[23] A. M. H. Gerards. 1995. Matching. In *Network Models*, M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser (Eds.). Elsevier, Amsterdam, 135–224.

[24] D. Harel and R. E. Tarjan. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.* 13, 2 (1984), 338–355.

[25] H. W. Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Res. Logist. Quart.* 2 (1955), 83–97.

[26] H. W. Kuhn. 1956. Variants of the Hungarian method for assignment problems. *Naval Res. Logist. Quart.* 3 (1956), 253–258.

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

[28] L. Lovász and M. D. Plummer. 1986. *Matching Theory*. North-Holland, New York.

[29] J. B. Orlin. 1993. A faster strongly polynomial minimum cost flow algorithm. *Op. Res.* 41 (1993), 338–350.
[30] C. H. Papadimitriou and K. Steiglitz. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Englewood Cliffs, NJ.
[31] S. Pettie. 2005. Sensitivity analysis of minimum spanning trees in sub-inverse-Ackermann time. In *Proceedings of the 16th International Symposium on Algorithms and Computation (LNCS 3827)*, X. Deng and D. Du (Eds.). Springer-Verlag, Berlin, 964–973.
[32] W. R. Pulleyblank. 1973. *Faces of Matching Polyhedra*. Ph.D. Dissertation. Department of Combinatorics and Optimization, Univ. of Waterloo.
[33] W. R. Pulleyblank. 2012. Edmonds, matching and the birth of polyhedral combinatorics. *Documenta Mathematica* (2012), 181–197.
[34] A. Schrijver. 2003. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, New York.
[35] R. E. Tarjan. 1979. Applications of path compression on balanced trees. *J. ACM* 26, 4 (1979), 690–715.
[36] R. E. Tarjan. 1983. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA.
[37] M. Thorup. 1999. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM* 46, 3 (1999), 362–394.
[38] G. M. Weber. 1981. Sensitivity analysis of optimal matchings. *Networks* 11 (1981), 41–56.

Received August 2016; accepted January 2018