Faster Scaling Algorithms for General Graph-Matching Problems

HAROLD N. GABOW

University of Colorado, Boulder, Colorado

AND

ROBERT E. TARJAN

Princeton University, Princeton, NJ, and NEC Research Institute, Princeton, NJ

Abstract. An algorithm for minimum-cost matching on a general graph with integral edge costs is presented. The algorithm runs in time close to the fastest known bound for maximum-cardinality matching. Specifically, let n, m, and N denote the number of vertices, number of edges, and largest magnitude of a cost, respectively. The best known time bound for maximum-cardinality matching is $O(\sqrt{n}\,m)$. The new algorithm for minimum-cost matching has time bound $O(\sqrt{n\alpha(m, n)}\log n\,m \log(nN))$. A slight modification of the new algorithm finds a maximum-cardinality matching in $O(\sqrt{n}\,m)$ time. Other applications of the new algorithm are given, including an efficient implementation of Christofides' travelling salesman approximation algorithm and efficient solutions to update problems that require the linear programming duals for matching.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Non-numerical Algorithms and Problems—*Computations on discrete structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms, network problems.*

General Terms: Algorithms, Design, Theory8GAugmenting path, blossom, matching, network optimization, scaling.

1. *Introduction*

Finding a minimum-cost matching on a general graph is a classic problem in network optimization. It has many practical applications and very efficient algorithms. We present an algorithm for this problem that is almost as fast as

Harold Gabow's research was supported in part by NSF Grant Nos. DCR-851191, CCR-8815636, and AT&T Bell Laboratories.

Robert Tarjan's research at Princeton University was supported in part by NSF Grant No. DCR-8605962 and ONR Contract No. N00014-87-K-0467. His research was also supported in part by AT&T Bell Laboratories.

Authors' addresses: Harold N. Gabow, Department of Computer Science, University of Colorado, Boulder, CO 80309; Robert E. Tarjan, Computer Science Department, Princeton University, Princeton, NJ 08544/NEC Research Institute, Princeton, NJ 08540.

the best known algorithm for the problem without costs, that is, maximum-cardinality matching.

In stating resource bounds, we use n and m throughout this paper to denote the number of vertices and the number of edges in the given graph, respectively. When the graph has integral-valued edge costs, N denotes the largest magnitude of a cost.

The fastest known algorithm for maximum-cardinality matching is due to Micali and Vazirani [23] and runs in time $O(\sqrt{n}\, m)$ (see also [14]). Edmonds gave the first polynomial algorithm for weighted matching [7]. At the time of the initial writing of this paper, the best known implementation of this algorithm was that of [13]; it runs in time $O(n(m \log\log\log_d n + n \log n))$, where $d = \max\{m/n, 2\}$ is the density of the graph. Recently an implementation achieving time $O(n(m \mp n \log n))$ has been given [12]. These bounds for weighted matching can be substantially improved under the assumption of integral costs that are not huge: The scaling algorithm of [10] runs in time $O(n^{3/4} m \log N)$. We improve this last bound to $O(\sqrt{n\alpha(m, n)} \log n\, m \log(nN))$. We also show that for maximum-cardinality matching our algorithm runs in the same time as the algorithm of Micali and Vazirani cited above. We present two other applications: We show how to speed up Christofides' travelling salesman approximation algorithm [4] to time $O(n^{2.5}(\log n)^{1.5})$; this bound is independent of the size of the input numbers. We also show how to find the linear programming dual variables for matching, which are the basis of Edmonds' algorithm. This gives efficient solutions to various matching update problems. Some more recent applications of our algorithm are mentioned in the last section.

Our algorithm is based on the approach to scaling introduced by Goldberg and Tarjan for the minimum-cost flow problem [18, 19] and applied in [16] to the assignment problem. Historically, the first approaches to scaling computed an optimum solution at each of $\log N$ scales (e.g., [7a], [20], [9, 10]). The new method computes an approximate optimum solution at each of $\log(nN)$ scales; using $\log n$ extra scales ensures that the last approximate optimum is exact. The notion of ϵ-optimality [3, 25] turns out to be the appropriate definition of "approximate optimum" for this scaling technique.

Applying this scaling technique to general graphs is difficult because of "blossoms." In all of the scaling algorithms mentioned above for bipartite and directed graphs, the solution to one scale gives an obvious starting point for the solution to the next. Blossoms invalidate the obvious starting point. The techniques of [10], including the notion of "shells," are used to overcome this difficulty. Nonetheless, blossoms slow our algorithm down: The algorithm of [16] finds a minimum-cost matching on a bipartite graph in time $O(\sqrt{n}\, m \log(nN))$. The extra factor of $\sqrt{\log n}$ in our bound for general matching comes from errors introduced in finding the starting point; the extra factor of $\sqrt{\alpha(m, n)}$ comes from data structures for blossom manipulation.

The paper is organized as follows. Section 1.1 reviews Edmonds' weighted matching algorithm [7]; many of the ideas and routines of this algorithm are incorporated into ours. The rest of the paper presents our algorithm in a top-down fashion. Section 2 gives the main routine, Sections 3–4 give lower level subroutines. These sections also show that the algorithm is correct and give parts of the efficiency analysis. Sections 5–6 essentially complete the

efficiency analysis. Sections 7–8 give the remaining lower-level details of the algorithm. Section 9 concludes the analysis of the algorithm. Section 10 applies the algorithm to other matching problems such as minimum perfect matching. Section 11 mentions a few further applications of the algorithm.

The present section closes with notation and definitions. We use several standard mathematical conventions to simplify the efficiency analysis. Background concerning matching can be found in greater detail in [21], [22], and [27].

If S is a set and e an element, $S + e$ denotes $S \cup \{e\}$ and $S - e$ denotes $S - \{e\}$. For integers i and j, $[i\,..\,j] = \{k \mid k \text{ is an integer, } i \le k \le j\}$. The function $\log n$ denotes logarithm to the base two.

We use a hat, for example, $\hat{f}$, to emphasize that an object is a function, when necessary. We use a dot, $\cdot$, to denote the argument of a function. For example, if f is a function of two variables, $f(x, \cdot)$ denotes the function of one variable mapping y to $f(x, y)$. If f and g are real-valued functions then $f + g$ and fg denote their sum and product, respectively, that is, $(f + g)(x) = f(x) + g(x)$, $fg(x) = f(x) \times g(x)$. We use the following conventions to sum the values of a function: If f is a real-valued function whose domain includes the set S, then $f(S) = \sum\{f(s) \mid s \in S\}$. Similarly if f has two arguments then $f(S, T) = \sum\{f(s, t) \mid s \in S,\ t \in T\}$, for $S \times T$ a subset of the domain of f.

For any graph G, $V(G)$, and $E(G)$ denote the vertex set and edge set of G, respectively. All graphs in this paper are undirected. We regard an edge as being a set of two vertices; hence, a statement like $e \subseteq S$, for e an edge and S a set of vertices, means that both vertices of e are in S. We usually denote the edge joining vertices v and w by vw. Thus if $e = vw$ and $y\colon E(G) \to \mathbf{R}$ then $y(e) = y(v) + y(w)$ by our convention for functions. We often identify a subgraph H, such as a path or tree, with its set of vertices $V(H)$ or edges $E(H)$. For example $H \subseteq S$ is short for $V(H) \subseteq S$ or $E(H) \subseteq S$, depending on whether S is a set of vertices or edges; the choice will be clear from the context.

A *matching* on a graph is a set of vertex-disjoint edges. Thus a vertex v is in at most one matched edge vv'; a *free* vertex is in no such edge. A *perfect* matching has no free vertices. An *alternating path* (*cycle*) for a matching is a simple path (cycle) whose edges are alternately matched and unmatched. 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 $M \oplus P$, thus giving a matching with one more edge ($M \oplus P$ denotes the symmetric difference of M and P).

Suppose $c\colon E \to \mathbf{R}$ is a function that assigns a numeric *cost* to each edge; in this paper costs are integers in $[-N \cdots N]$ unless stated otherwise. By our convention the cost $c(S)$ of a set of edges S is the sum of the individual edge costs. A *minimum-* (*maximum-*) *cost matching* is a matching of smallest (largest) possible cost. A *minimum* (*maximum*) *perfect matching* is a perfect matching of smallest (largest) possible cost.

1.1 Edmonds' Minimum Critical Matching Algorithm. It is convenient to work with a variant of the matching problem defined as follows. Let G be a graph and v a fixed vertex. A *v-matching* is a perfect matching on $G - v$. Figure 1 shows an x-matching; in all figures of this paper matched edges are drawn wavy and free vertices are drawn square. A *minimum* (*maximum*)

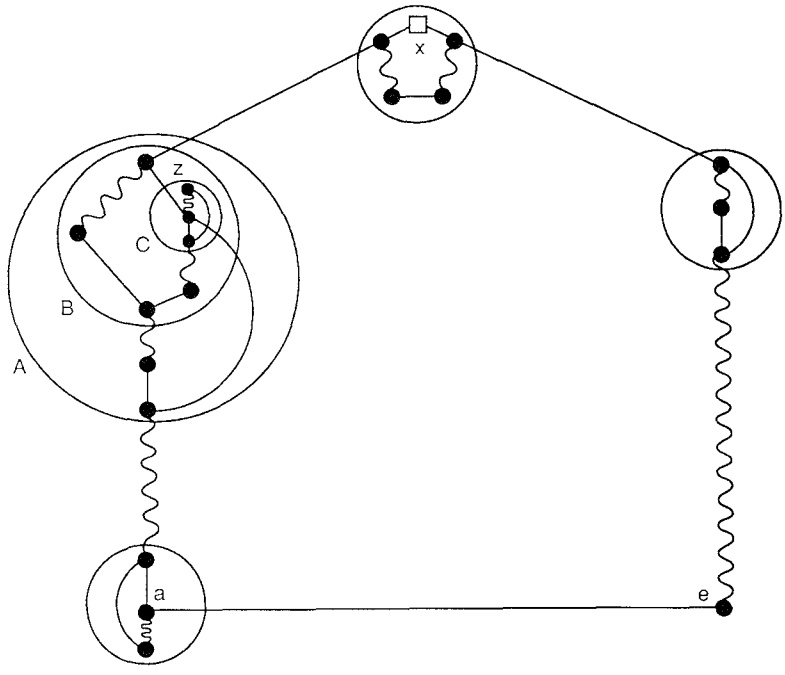

FIG. 1. Blossom with base vertex x

v-matching is a v-matching with minimum (maximum) possible cost. G is *critical* if every vertex v has a v-matching. The *minimum critical matching problem* is, given a critical graph, to find a minimum v-matching for each vertex v [11]. It follows from [6] that all the desired matchings can be represented by the blossom tree defined below; we take this blossom tree as a solution to the critical matching problem.

Note that if G is a graph with a perfect matching, a critical graph is obtained by adding a vertex adjacent to every vertex of $V(G)$. Hence, an algorithm for minimum critical matching can be used to find a minimum perfect matching.

Edmonds' algorithm is based on the notion of blossom, which is explained in the next four paragraphs. Let G be a graph with a matching. A *blossom forest* F is a forest with the following properties. (Figure 2 shows a blossom forest, with just one tree, for Figure 1.) The number of children of any nonleaf node of F is at least three and odd. Each node of F is identified with a subgraph of G, as follows: The leaves of F are precisely the vertices of G. If B is a nonleaf node, its children can be ordered as B_i, $i = 1, \ldots, k$ so that

$$V(B) = \bigcup_{i=1}^{k} V(B_i), \quad E(B) = \bigcup_{i=1}^{k} \left(E(B_i) + e_i\right),$$

where e_i is an edge that joins a vertex of $V(B_i)$ to a vertex of $V(B_{i+1})$ (interpret B_{k+1} as B_1); furthermore e_i is matched precisely when i is even. (Thus, each child of B is incident to two edges e_i; for B_1 both edges are unmatched, and for all other children one edge is matched and the other unmatched; there are precisely two possible orderings for the children.) In this

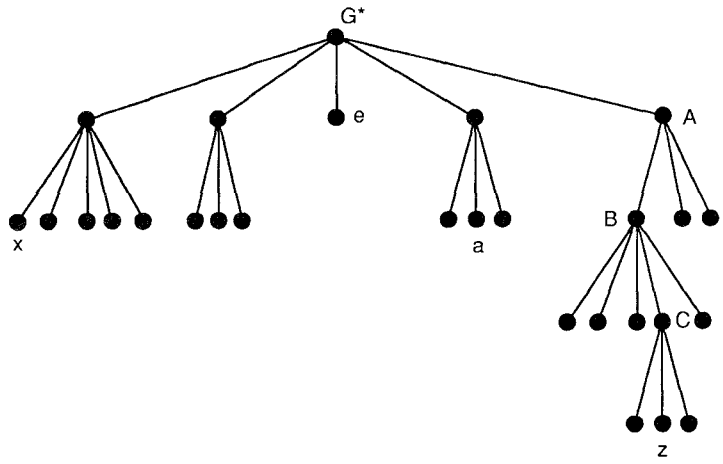

FIG. 2. Blossom tree.

paper, *node* always refers to an element of $V(F)$ and *vertex* always refers to an element of $V(G)$.

Each node B of F is a *blossom*. (Thus, a blossom can also be regarded as a subgraph.) The *blossom edges* of B are the above edges e_i, $i = 1, \ldots, k$. Any root, that is, maximal blossom, is a *root blossom*; all other blossoms are *nonroot blossoms*. Every vertex is a blossom; a blossom that is not a vertex is a *nonleaf blossom*.

The subgraph *induced* by $V(B)$ is denoted $G(B)$. Define functions

$$\hat{n}(B) = |V(B)|, \quad \hat{m}(B) = |E(G(B))|.$$

We emphasize that a blossom is not defined as an induced subgraph, e.g., $\hat{m}(B)$ is in general larger than $|E(B)|$. A simple induction shows that $\hat{n}(B)$ is odd. The *base vertex* of B is the unique vertex of B not in a matched edge of $E(G(B))$. The base of a vertex v is v; a simple induction shows that the base of a nonleaf blossom B exists and is in the first child blossom B_1 of B.

Any v-matching of a critical graph has a blossom forest that consists of one tree T^* called a *blossom tree*. (This can be proved by examining the algorithm of [6].) The root of T^* is a blossom having vertex set $V(G)$ and is denoted G^*. Given T^*, for any vertex w a w-matching of G can be found in time $O(n)$. We now describe a recursive procedure to do this. The procedure is *blossom_match*(B, w); here, B is a nonleaf node of T^*, w is a vertex of B, and the procedure constructs a w-matching of B. To do this, let B have children B_i and blossom edges e_i, $i = 1, \ldots, k$; as above, e_i joins B_i to B_{i+1}. Let $w \in B_j$. Match alternate edges of the list e_i, keeping e_{j-1} and e_j unmatched. For $i \neq j$ let w_i denote the vertex of B_i on a newly matched edge, and let $w_j = w$. Complete the procedure by recursively executing *blossom_match*(B_i, w_i) for each nonleaf child B_i. It is easy to see that *blossom_match*(T^*, w) constructs the desired w-matching in time $O(n)$.

Now we review Edmonds' algorithm for minimum critical matching; further details can be found in [7]. Two functions y, z form (a pair of) *dual functions* if $y: V(G) \to \mathbf{R}$, $z: 2^{V(G)} \to \mathbf{R}$ and $z(B) \geq 0$ unless $B = V(G)$. Such a pair

determines a *dual edge function* yz: $E \to \mathbf{R}$, which for an edge e is defined as

$$yz(e) = y(e) - z(\{B \mid e \subseteq B\}).$$

(Recall that by convention if $e = vw$ then $y(e) = y(v) + y(w)$. Similarly, the last term denotes $\sum\{z(B) \mid e \subseteq B\}$.) The duals are *dominated* on edge e if

$$yz(e) \leq c(e);$$

they are *tight* if equality holds.

Edmonds' algorithm maintains a *structured matching*. This is a matching plus corresponding blossom forest F plus dual functions that collectively satisfy two conditions: (i) z is nonzero only on nonleaf blossoms of F. (ii) The duals are dominated on every edge, and tight on every edge that is matched or a blossom edge. A structured v-matching is a minimum v-matching. (This can be proved by an argument similar to Lemma 2.1(a) below.) Regarding (i), define a *weighted blossom* as a blossom with a nonzero dual.

An *optimum structured matching* is a structured v-matching for some vertex v, whose blossom forest is a blossom tree T^*. Given T^*, for any vertex w a minimum w-matching is found in $O(n)$ time by the *blossom_match* procedure. The output of Edmonds' algorithm is an optimum structured matching. Thus, Edmonds' algorithm solves the minimum critical matching problem.

The input to Edmonds' algorithm is a critical graph plus a structured matching. (The structured matching can be the empty matching, a blossom forest of isolated vertices, and dual functions $z = 0$ and y small enough to be dominated on every edge.) The algorithm repeatedly does a "search" followed by an "augment step" until some search halts with a v-matching (v arbitrary) and a blossom tree (not forest). (This is a slight difference from the way the algorithm of [7] halts; see below.)

More precisely a *search* builds a *search graph* $\mathcal{S}$, defined as follows and illustrated in Figure 3. $V(\mathcal{S})$ is partitioned into root blossoms B. $E(\mathcal{S})$ consists of the blossom edges $E(B)$ plus other tight edges. The rest of the description of $\mathcal{S}$ depends on whether or not an augmenting path has been found. First consider $\mathcal{S}$ before an augmenting path has been found. If each root blossom of $\mathcal{S}$ is contracted to a vertex, $\mathcal{S}$ becomes a forest $\mathcal{F}$ ($\mathcal{F}$ should not be confused with the blossom forest). The roots of $\mathcal{F}$ are precisely the root blossoms of G that contain a free vertex. The path $P(B)$ from a root blossom B to the root of its tree in $\mathcal{F}$ is alternating. Blossom B is *outer* (*inner*) if $P(B)$ has even (odd) length. Any descendant, in the blossom forest, of an outer (inner) root blossom is also called outer (inner). (In particular, every free vertex of G is outer.) Any outer vertex v is joined to a free vertex by an even-length alternating path $P(v) \subseteq E(\mathcal{S})$; if v is in root blossom B then $P(v)$ contains $P(B)$. For instance Figure 3 shows path $P(a)$ for vertex a of Figure 1.

Now consider $\mathcal{S}$ when an augmenting path has been found. In this case, $\mathcal{S}$ contains one or more tight edges vw joining outer vertices in distinct trees of $\mathcal{F}$. Each such edge gives an augmenting path composed of vw plus the above paths $P(v)$, $P(w)$.

The search builds $\mathcal{S}$ using three types of steps. A *grow step* enlarges $\mathcal{S}$ by adding a tight edge e that was incident to $\mathcal{S}$; the root blossom B at the end of e is also added to $\mathcal{S}$. Grow steps always occur in pairs in Edmonds' algorithm:

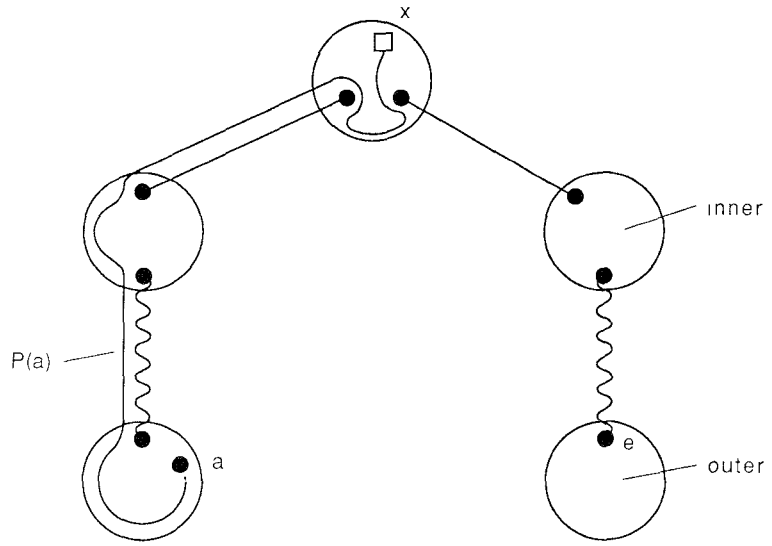

FIG. 3 Search graph in Edmonds' algorithm.

first an unmatched edge e is added, along with the above blossom B; then, the matched edge incident to B is added. Figure 4 shows a grow step for the unmatched edge ab followed by a grow step for the matched edge cd.

A *blossom step* enlarges $\mathcal{F}$ by adding a tight edge that joins distinct outer root blossoms of $\mathcal{F}$. This step either constructs a new blossom in $\mathcal{F}$, or it discovers that $\mathcal{F}$ contains an augmenting path. In Figure 3 an edge ae would give a blossom step that constructs a new blossom, possibly the one in Figure 1.

An *expand step* deletes an unweighted root blossom B from the blossom forest, thus making its children into roots; B is also deleted from $\mathcal{F}$ and replaced by some of these children (not necessarily all of them) so that $\mathcal{F}$ remains a forest. Figure 5 shows an expand step; blossoms $B_1, \ldots, B_5$ become root blossoms, and B_4 and B_5 leave $\mathcal{F}$.

These three steps are repeated as many times as possible, giving a maximal search graph $\mathcal{F}$. If the maximal $\mathcal{F}$ does not contain an augmenting path and G is not a blossom, a *dual adjustment* is done. It starts by computing a quantity δ, as described below. Then, it makes the following changes:

$$
\begin{aligned}
&y(v) \leftarrow y(v) + \delta, && \text{for each outer vertex } v;\\
&y(v) \leftarrow y(v) - \delta, && \text{for each inner vertex } v;\\
&z(B) \leftarrow z(B) + 2\delta, && \text{for each nonleaf root outer blossom } B;\\
&z(B) \leftarrow z(B) - 2\delta, && \text{for each nonleaf root inner blossom } B.
\end{aligned}
$$

These assignments do not change the value of $yz(e)$ for $e \in E(\mathcal{F})$, so these edges remain tight. The assignments increase $yz(e)$ only if e joins an outer vertex to a vertex not in $\mathcal{F}$, or if e joins two distinct root outer blossoms. Thus the adjustment maintains condition (ii) above and also allows a new grow,

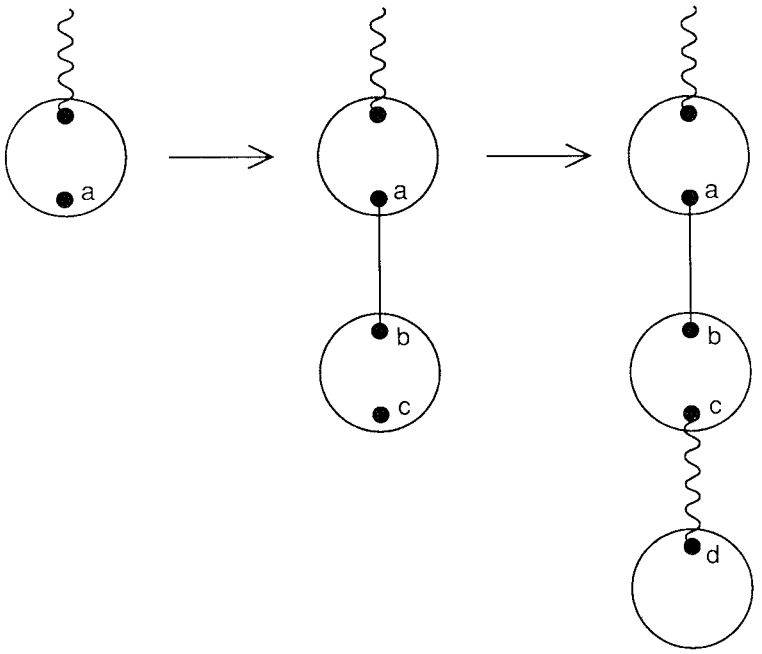

FIG. 4. Grow steps in Edmonds' algorithm

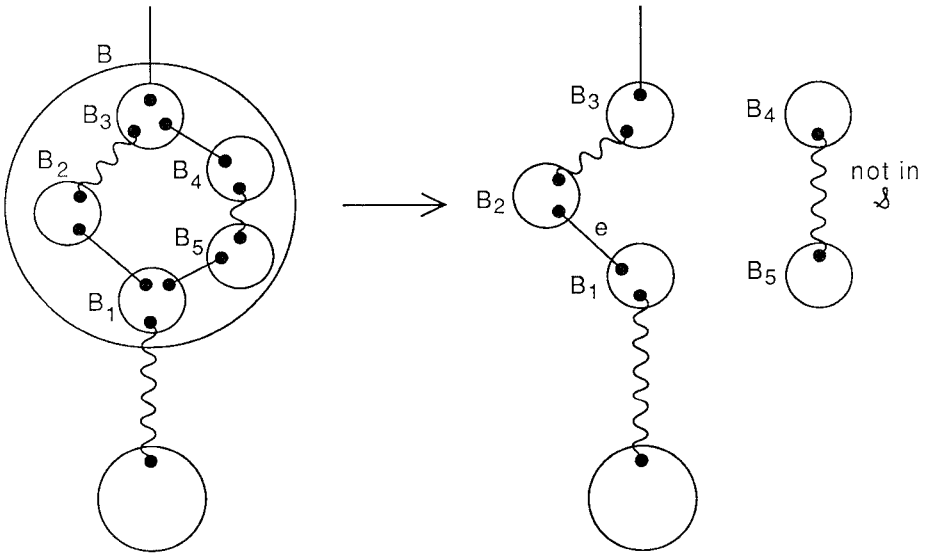

FIG. 5. Expand step.

blossom or expand step to be executed, if δ is chosen as $\min\{\delta_g, \delta_b, \delta_e\}$, where

$$\delta_g = \min\{(c - y)(vw) \mid vw \in E(G),\ v \text{ an outer vertex},\ w \notin V(\mathcal{S})\};$$

$$\delta_b = \min\left\{\frac{(c - y)(e)}{2} \,\middle|\, e \text{ an edge joining two distinct root outer blossoms}\right\};$$

$$\delta_e = \min\left\{\frac{z(B)}{2} \,\middle|\, B \text{ a nonleaf root inner blossom}\right\}.$$

Note that $\delta > 0$ by the maximality of $\mathcal{S}$. If $\delta = \delta_g$, a grow step can be executed after the dual adjustment; similarly, $\delta = \delta_b$ gives a blossom step and $\delta = \delta_e$ gives an expand step.

After the dual adjustment the search continues to do grow, blossom and expand steps. Eventually the search halts, in one of two ways. Every search but the last finds a *weighted augmenting path*. This is an augmenting path P whose edges are tight. The *augment step* enlarges the matching M by one edge to $M \oplus P$. The blossom forest and duals remain valid. Then the algorithm continues with the next search. In the last search the matching is a v-matching. The last search eventually absorbs the entire graph G into one blossom. At this time, the algorithm halts with the desired optimum structured matching.

We note two more properties of Edmonds' algorithm for use below. First, any dual adjustment increases the y value of any free vertex by δ. Second, note that a dual adjustment step does divisions by two to calculate δ_b, δ_e. If all given costs are even integers, and all y values are initialized to have the same parity, then all quantities computed by the algorithm are integers [24, p. 267, ex. 3]. This integrality property motivates various details of the scaling algorithm. For instance, the scaling algorithm keeps all edge costs even.

It will be helpful to sketch a proof of the above integrality property. Suppose that all given costs are even integers and all y values are initialized to integers of the same parity. The algorithm maintains the invariant that all y values are integral, all y values of free vertices have the same parity, and all z values are even integers. The main step of the proof is to observe that all y values of vertices in $\mathcal{S}$ have the same parity. This follows since $\mathcal{S}$ consists of tight edges and each connected component contains a free vertex. Using this observation it is easy to see that the invariant is preserved.

2. *The Matching Algorithm: The Scaling Routine*

This section gives the overall structure of our matching algorithm. This algorithm solves the minimum critical matching problem in $O(\sqrt{n\alpha(m, n)}\log n\ m \log(nN))$ time. This section describes the main routine of the scaling algorithm for minimum critical matching. The input is a critical graph. (This entails no loss of generality—the algorithm can detect input graphs that are not critical, as indicated below.)

The algorithm works by scaling the costs. Each scale finds a v-matching, for some v, whose cost is almost minimum in the following sense. A *2-optimum matching* is a v-matching M_v, for some vertex v, plus a blossom tree T, plus dual functions y, z such that z is nonzero only on nonleaves of T and the

following constraints hold:

$$yz(e) \le c(e), \qquad \text{for} \quad e \in E; \tag{1a}$$

$$yz(e) \ge c(e) - 2, \qquad \text{for} \quad e \in M \cup \bigcup \{E(B) \mid B \in V(T)\}. \tag{1b}$$

Note that if this definition is satisfied for some vertex v, it is satisfied by every vertex x (matching M_x is constructed in $O(n)$ time by the *blossom_match* procedure). Hence, when M_x denotes a 2-optimum matching, we understand that x can be chosen arbitrarily.

To motivate this definition, first observe that dropping the -2 term from (1b) gives the dominated and tight conditions used in Edmonds' algorithm. The -2 term is included so that the algorithm augments along paths of short length. This makes the algorithm efficient. Further motivation is given in [16]. (Actually the bipartite matching algorithm of [16] uses a term of magnitude 1 rather than 2, and also maintains equality in the constraint for matched edges. Here, we use magnitude 2 because of the aforementioned considerations of integrality. Also equality cannot be maintained on the matched edges, because of details of blossom manipulation.) The following result is the analog of Edmonds' optimality condition.

LEMMA 2.1. *Let M_x be a 2-optimum matching.*

(*a*) *For any vertex x, any x-matching X has $c(X) > c(M_x) - n$.*
(*b*) *If each cost $c(e)$ is a multiple of some integer k, $k \ge n$, then M_x is a minimum x-matching.*

PROOF

(a) Consider any vertex x and let T be the blossom tree. Function z is nonzero only on nonleaves of T. For any blossom B, M_x contains precisely $\lfloor \hat{n}(B)/2 \rfloor$ edges of $G(B)$, and no matching contains more. Combining these facts with (1a) and (1b) gives

$$\begin{aligned} c(M_x) &\le 2\lfloor n/2 \rfloor + yz(M_x) \\ &\le (n-1) + y(V(G) - x) - \lfloor \hat{n}/2 \rfloor z(V(T)) \le c(X) + n - 1. \end{aligned}$$

(Recall that by the conventions of Section 1, $\lfloor \hat{n}/2 \rfloor z(V(T))$ denotes $\sum\{\lfloor \hat{n}(B)/2 \rfloor z(B) \mid B \in V(T)\}$.)

(b) This follows from (a) and the fact that any matching has cost a multiple of k. □

Now we describe the *scaling routine*, the main routine of the algorithm. It scales the costs in the following manner. The algorithm always works with even edge costs to preserve integrality. Initially it computes a new cost function $\bar{c}$ equal to $n + 1$ times the given cost function. (Clearly the two cost functions have the same minimum x-matchings. Also each $\bar{c}(e)$ is even.) It maintains a cost function c equal to $\bar{c}$ in the current scale. More precisely define $k = \lfloor \log(n+1)N \rfloor + 1$, the greatest number of bits in the binary expansion of a $\bar{c}$ cost. For any $s \in [1 \cdots k]$ define a function $b_s: [-(n+1)N \cdots (n+1)N] \to \{-1, 0, 1\}$ by taking $b_s(i)$ as the sth signed bit in the expansion of i as a k-bit binary number. Thus, any new cost value $i = \bar{c}(e)$ equals $\sum_{t=1}^{k-1} b_t(i) 2^{k-t}$ (since $b_k(i) = 0$). In the sth scale, the cost $c(e)$ is taken to be $\sum_{t=1}^{s} b_t(i) 2^{s+1-t}$, so in the $(k-1)$st scale $c(e) = i$.

The *scaling routine* computes $\bar{c}$, then initializes c, y and z to the zero function, the matching M_x to $\varnothing$, and the blossom tree T to a root G with children $V(G)$. Then, it executes the following loop for index s going from 1 to $k-1$:

Double Step. Compute new functions $c \leftarrow 2(c + b_s)$, $y \leftarrow 2y - 1$, $z \leftarrow 2z$.

Match Step. Call the *match* routine to find a 2-optimum matching M_x, with new duals y, z and new blossom tree T.

Lemma 2.1(b) implies that if *match* works as described in the Match Step, the *scaling routine* solves the minimum critical matching problem, that is, each final matching M_x is a minimum x-matching. Each iteration of the loop is called a *scale*. We give a *match* routine that runs in $O(\sqrt{n\alpha(m, n)}\log n\, m)$ time, thereby achieving the desired time bound.

Note that in the first scale the tree T computed in the initialization is not necessarily a blossom tree, since it need not correspond to a blossom structure. We shall see that the algorithm still works correctly, because $z = 0$ (see the last paragraph of Section 4).

3. *The Match Routine*

This section describes the overall structure of the routine that finds a 2-optimum matching in a scale.

On entry to *match*, y, z are duals computed in the Double Step, and T is the blossom tree of the previous scale (or the initialization, in the first scale). The *match* routine saves T as the tree T_0; for the analysis it is convenient to let y_0, z_0 refer to the duals on entry to *match*.

The *scaling routine* is similar to the main routine of the bipartite matching algorithm of [16]. In the bipartite algorithm, each scale is similar to the first in that the dual function can be initially taken as zero, and there is no structure on the graph inherited from previous scales. This is not true for general graphs: The function z_0 can have positive values that cannot be eliminated (see [10]). The *match* routine is forced to work with blossoms from both the previous scale, in blossom tree T_0, and the current scale. It is convenient to denote the current blossom forest as T (eventually this forest becomes a blossom tree). An *old blossom* is a node of $V(T_0)$; a *current blossom* is a node of $V(T)$. An old blossom B *dissolves* either when it becomes a current blossom or, if $B \neq G^*$, when $z(B)$ becomes zero. Note that current blossoms do not dissolve (in the current scale); hence, we use the term *undissolved blossom* to refer to an old blossom that has not yet dissolved. A vertex is a current blossom, so only nonleaf blossoms are undissolved. Finally, note that the old matching is implicitly discarded in the Double Step, so "the matching" refers to the current matching.

The *match* routine maintains inequalities (1), with z nonzero only on nonleaves of $V(T) \cup V(T_0)$. In (1b) T is the current blossom forest. Observe that both current and old blossoms contribute to the z term in the definition of $yz(e)$. When all old blossoms are dissolved, the matching is 2-optimum and the routine can halt. The reason is that by definition the old blossom G^* can dissolve only by becoming a current blossom. When this occurs we have a v-matching, a blossom tree, and a function z that is nonzero only on nonleaves of T.

Note that after the Double Step, (1a) holds for all edges and (1b) is vacuous. Hence, the Double Step maintains (1) as desired. To help preserve (1a) the *match* routine also maintains

$$y \leq y_0. \tag{2}$$

In a blossom tree, define the *major child* C of a node B as a child with largest size $\hat{n}(C)$; a tie for the major child is broken arbitrarily. Hence, any nonleaf has exactly one major child, and any nonmajor child D of B has $\hat{n}(D) < \hat{n}(B)/2$. A *major path* is a maximal path in which each node is followed by its major child. The major paths partition the nodes of a blossom tree. (These are essentially the "heavy paths" of [26].) A major path starting at node R is denoted by $P(R)$ and has *major path root* R. Define the *rank* of any node B as $\lfloor \log \hat{n}(B) \rfloor$. A nonmajor child of a node B has rank less than B; a nonmajor child of a node in $P(R)$ has rank less than R. In Figure 2 the path from the root to leaf z is the major path $P(G^*)$; the root has rank $\lfloor \log 21 \rfloor = 4$.

procedure *match*.
Initialize the matching to $\varnothing$ and T to the forest having every vertex of G a root. Traverse the major path roots R of T_0 in postorder. At each root R call a routine *path*(R) to dissolve the old blossoms on $P(R)$, while maintaining (1)–(2).

This routine is correct since after *match* processes root $R = G^*$, all old blossoms are dissolved. Thus, the matching is 2-optimum as observed above. Note that for any major root R, on entry to *path*(R) all descendants of R have dissolved except those on $P(R)$. Figure 6 illustrates this situation: Suppose the graph of Figure 1 has old blossom tree given by Figure 2. Then, on entry to *path*(G^*), all blossoms are dissolved except those shown on $P(G^*)$.

LEMMA 3.1. *If the time for path*(R) *is* $O(\sqrt{\hat{n}(R)\alpha(m, n)\log n}\,\hat{m}(R))$, *then the time for match is* $O(\sqrt{n\alpha(m, n)\log n}\,m)$.

PROOF. For any integer $0 \leq r \leq \log n$, consider the major path roots of rank r. For any vertex $v \in V(G)$, at most one of these roots R has $v \in V(R)$. Hence any edge (of $E(G)$) is in at most one of the subgraphs $G(R)$. Thus for some constant c the time spent on these roots is at most $c\sqrt{2^{r+1}\alpha(m, n)\log n}\,m$. Summing over all ranks r gives the desired bound. □

4. *Shells and the Path Routine*

This section presents the *path* routine and its main subroutine *shell_search*. These routines are based on the concept of a shell [10].

If C and D are blossoms with $V(D) \subseteq V(C)$, the *shell* $G(C, D)$ is the subgraph induced by $V(C) - V(D)$. C is the *outer boundary*, D the *inner boundary*. As a special case, we allow $D = \varnothing$. Extend the function $\hat{n}$ to shells: $\hat{n}(C, D)$ is the number of vertices in a shell $G(C, D)$. A shell is *even* if $\hat{n}(C, D)$ is even, or equivalently $D \neq \varnothing$; otherwise, it is *odd*. Figure 6 indicates the even shell $G(G^*, A)$.

We use a number of functions of shells, such as the above $\hat{n}$. We define such functions by using the shell boundaries as arguments, as in the above $\hat{n}(C, D)$. Alternatively, if X denotes a shell, we use X as the argument, e.g., $\hat{n}(X)$. If

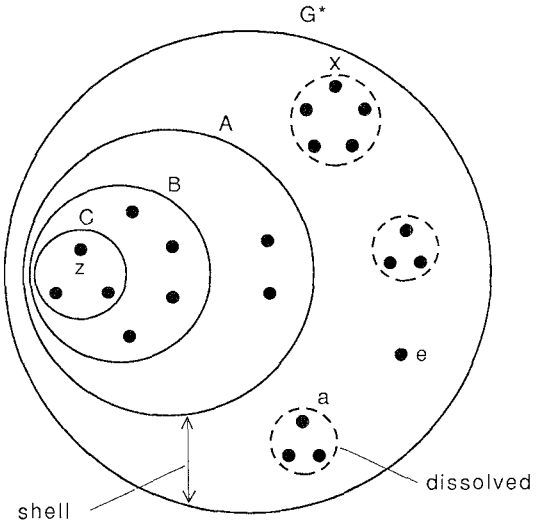

FIG. 6. Major path with dissolved blossoms.

X denotes an old blossom, it corresponds to an odd shell, and we write $\hat{n}(X)$ as a shorthand for $\hat{n}(X, \emptyset)$. Which of the two interpretations of $\hat{n}(X)$ is appropriate will always be clear from context.

This paper only refers to *shells of* $P(R)$, which are shells $G(C, D)$ with C, D on $P(R)$ for the major path root R (D may be empty). At any time in *path*(R), if C and D are currently consecutive undissolved blossoms in $P(R)$, then $G(C, D)$ is an *undissolved shell* (*of* $P(R)$). (An undissolved odd shell has C the currently innermost undissolved blossom of $P(R)$.) The *path*(R) routine works with undissolved shells. Obviously these shells change as blossoms dissolve.

The *path*(R) routine works like the bipartite matching algorithm of [16] in the sense that it finds most augmenting paths quickly and finds the remaining paths at a slower and slower rate. The bipartite algorithm accomplishes this automatically, that is, the algorithm is unchanging, only its performance changes. For general graphs, it seems that some lower-level details of the algorithm must change as the execution progresses. For this reason, we organize the *path* routine in "phases." More precisely, the *phase* is defined in terms of a parameter ρ whose value is chosen below (Section 5). Also define R' to be the largest undissolved blossom of $P(R)$. (Initially $R' = R$ if R is a weighted blossom; R' shrinks as the algorithm progresses.) The *path* routine is a loop. Routine *path* is in phase 1 during the first ρ iterations of the loop. After that, it is in phase 2 if R' has more than one free vertex, and phase 3 otherwise. (Hence, in phase 3, R' has exactly one free vertex.) It will be apparent that *path* can go through any sequence of phases that starts with phase 1 and never decreases, that is, 1; 1, 2; 1, 2, 3; or 1, 3. We shall also see that an individual iteration consumes more time in a higher phase than in a lower.

The *path*(R) routine augments the matching along paths of "eligible" edges; it finds these paths by constructing a search graph of eligible edges. Edge e is defined to be *eligible* if its vertices are in the same undissolved shell of $P(R)$;

furthermore a condition that depends on the phase is satisfied. In phase 1 the condition is that one of these alternatives holds:

(i) e is unmatched and $yz(e) = c(e)$;
(ii) e is matched and $yz(e) = c(e) - 2$;
(iii) e is a current blossom edge.

In phase 2 or 3 the condition is $\hat{yz}(e) = c(e)$ or $c(e) - 2$. Note that this is always the case if any of (i)–(iii) hold.

Observe a simple consequence of the definition of eligibility: Throughout *path*(R), any matched edge or current blossom edge has both ends in the same undissolved shell (recall that the matched or current blossom edges are precisely the edges referred to in inequality (1b)). This observation follows because an edge must be eligible in order to be on an augmenting path and become matched or to enter a search graph and become a blossom edge; an eligible edge has both ends in the same undissolved shell of $P(R)$; the latter property continues to hold even as blossoms subsequently dissolve.

Here is the *path* routine. It uses a routine *shell_search*(S) whose argument is a shell S. As above, R' denotes the largest undissolved blossom in $P(R)$.

procedure *path*(R).
Repeat the following steps until all old blossoms of $P(R)$ are dissolved:

Augment Step. Construct an auxiliary graph H from $G(R')$ by contracting every current root blossom in R' and keeping only the eligible edges of $G(R')$. Find a maximal set $\mathcal{P}$ of vertex-disjoint augmenting paths in H. For each path of $\mathcal{P}$, augment along the corresponding path in G

Sort Step. Order the undissolved shells of $P(R)$ that contain a free vertex as S_i, $i = 1, \ldots, k$ so that $\hat{n}(S_i)$ is nonincreasing.

Search Step. For $i = 1$ to k, if both boundaries of S_i are still undissolved call *shell_search*(S_i) to adjust duals and possibly find an augmenting path of eligible edges (more details of *shell_search* are given below).

The *path* routine is implemented as follows: It exits immediately if R is a leaf blossom (any vertex is dissolved). Otherwise, the Augment Step finds augmenting paths by doing a depth-first search on H. The details of this search are unimportant for the analysis and so are postponed until Section 8. It suffices to note that the Augment Step uses time $O(\hat{m}(R))$.

In the Search Step, the *shell_search* routine is a *search* of Edmonds' algorithm modified in three ways: (i) to use eligibility rather than tightness; (ii) to take old blossoms into account; (iii) to change the halting procedure. We discuss each of these in turn.

For (i), note that eligibility plays the role of tightness in Edmonds' algorithm: *shell_search* adds an edge to the search graph $\mathcal{S}$ only when it is eligible. However, a matched edge need not be eligible. (This occurs only in phase 1.) Thus, a grow step may not be done for a matched edge incident to $\mathcal{S}$; also a blossom step may be done when a matched edge is scanned. (For example, in Figure 4, the matched edge cd may be added to $\mathcal{S}$ in a grow step that does not immediately follow the one for ab.) This contrasts with Edmonds' algorithm, where a matched edge is always tight, a grow step is always done for a matched

edge incident to $\mathcal{S}$, and a blossom step is done only when an unmatched edge is scanned. These changes to *search* are straightforward.

We turn to (ii). Consider an undissolved blossom B. To *translate B by* δ means to perform the following assignments:

$$y(v) \leftarrow y(v) - \delta, \quad \text{for each } v \in V(B);$$
$$z(B) \leftarrow z(B) - 2\delta.$$

(For example, a dual adjustment in Edmonds' algorithm translates inner root blossoms.) Observe that translating an undissolved blossom B cannot increase a quantity $yz(e)$, and it maintains inequalities (1) ((1b) holds since as observed above, no matched or current blossom edge has exactly one vertex in the undissolved blossom B).

Consider an undissolved shell $G(C, D)$ containing a free vertex. The routine *shell_search*(C, D) executes a *search* of Edmonds' algorithm on $G(C, D)$, modified to translate C and D. More precisely, when *search* does a dual adjustment, it calculates $\delta = \min\{\delta_g, \delta_b, \delta_e, \delta_d\}$, where the first three quantities correspond to the calculation in Edmonds' algorithm (Section 1.1) and $\delta_d = \min\{z(C)/2, z(D)/2 \mid D \neq \emptyset\}$. In the dual adjustment, *shell_search* translates C by δ and also translates D by δ (if $D \neq \emptyset$). *Shell_search* does a *dissolve* step if $\delta = \delta_d$, that is, the translation dissolves C or D (or both). *Dissolve* enlarges the shell to $G(C', D')$, where C' is the smallest undissolved blossom containing C and D' is the largest undissolved blossom contained in D. (Possibly $C' = C$ or $D' = D$, but not both. If C' does not exist, the search halts, as discussed with (iii) below.) Any free vertex that gets added to the shell is immediately added to $\mathcal{S}$ as an outer vertex. After the *dissolve* step, the search continues, now working on the enlarged shell $G(C', D')$.

We now verify that translating C and D ensures that inequalities (2) and (1) are preserved. Translating C ensures (2) is preserved, since *search* increases a y-value by at most δ. Consider (1), for any edge e of G. If e is in the shell, then (1) follows easily from Edmonds' algorithm and the fact that the translation does not change $yz(e)$. For other edges e, only (1a) applies, so it suffices to show that $yz(e)$ does not increase. If neither end of e is in C or both ends of e are in D, then obviously $yz(e)$ does not change. If precisely one end of e is in C, then (2) implies that $yz(e)$ does not increase. The remaining case is one end v in $V(D)$ and the other end w in $V(C) - V(D)$. The translations decrease $y(v)$ by 2δ and $z(C)$ by 2δ; $y(w)$ increases by at most δ in Edmonds' search and decreases by δ in the translations; thus, $yz(e)$ does not increase.

Finally, we discuss (iii). Recall that a *search* of Edmonds' algorithm halts either when it finds an augmenting path or (in the last search) when the entire graph is a blossom. *Shell_search* also halts in these circumstances. (Note that if it finds an augmenting path P, then P consists of eligible edges. *Shell_search* can halt because the current shell is a blossom only for an odd shell.) In addition, *shell_search* can halt in several other circumstances. We discuss these for the three phases in turn.

In phase 1, each execution of *shell_search* does at most one dual adjustment, and this adjustment uses $\delta = 1$. (In fact, Lemma 5.1 below shows that each phase 1 execution of *shell_search* does precisely one dual adjustment.) After this adjustment, the search halts. This contrasts with Edmonds' algorithm,

where *search* chooses δ large enough so that $\mathcal{S}$ can change. An adjustment of $\delta = 1$ may not allow any changes (or any augments in the following Augment Step).

In phase 1, after the dual adjustment there may be unweighted current root blossoms. Specifically, an inner root blossom can become unweighted in the dual adjustment. *Shell_search* repeatedly removes any unweighted root of the current blossom forest and replaces it by its children, until every nonleaf root is weighted. (This accomplishes what an *expand* step would have done in Edmonds' algorithm. It is done for the same reason—an unweighted blossom may be hiding an augmenting path.) Because of this rule, in any phase 1 Augment Step after the first, any current root blossom is weighted.

In phase 2 *shell_search*(S) halts on entry if S is an odd shell that does not contain all the free vertices of R'. Also it halts when a *dissolve* step enlarges the shell so that other halting criteria apply. This can happen in three ways. First, *dissolve* may add to S a new shell S' that has already been searched (in the current Search Step) and found to contain an augmenting path. Second, *dissolve* may enlarge S to an odd shell not containing all free vertices of R'. The third possibility is implied by the fact that only undissolved shells of $P(R)$ are searched. Consider the undissolved shell $G(R', D)$, where D is the largest undissolved blossom in R'. If *shell_search* dissolves R', $G(R', D)$ is no longer contained in an undissolved shell of $P(R)$. So the search halts. (Note that any augmenting paths that have been created in *shell_search* will be processed with the major path containing the parent of R; this case never occurs when $R = G^*$.)

There are no extra halting rules for phase 3. In phase 3 by definition no augmenting path can be found entirely in R. Thus *shell_search* halts when all blossoms on $P(R)$ are dissolved.

This concludes the statement of *path*. We now summarize some facts about *shell_search* that further motivate and justify the phase structure; the details of *shell_search* are in Section 7.

As already mentioned, the Search Step consumes more time in later phases. The usual implementation of *search* in Edmonds' algorithm (e.g., [13]) uses a priority queue to find the next dual adjustment quantity δ. In phase 1 *search* can be implemented without a priority queue, since only one dual adjustment is made. The proper data structures make the time for one Search Step $O(\hat{m}(R))$. In phase 2 the priority queue can be implemented as an array. The phase 2 Search Steps collectively use total time $O(\hat{n}(R)\log \hat{n}(R))$ to scan the array; in addition, each Search Step uses $O(\hat{m}(R)\alpha(m, n))$ time. In phase 3 a standard priority queue is used; the time for phase 3 is (less than) $O(\hat{m}(R)\log \hat{n}(R))$.

Next we indicate why the criteria for eligibility change for phases 2–3. Recall that in *search*, if a dual adjustment makes an inner blossom B unweighted an *expand* step is done. As illustrated in Figure 5, *expand* replaces B in $\mathcal{S}$ by an alternating path of edges of $E(B)$. *Expand* steps do not occur in phase 1, since a phase 1 search stops right after its dual adjustment. *Expands* do occur in phases 2–3. Now observe that an edge $e \in E(B)$ that *expand* places in $\mathcal{S}$ may not satisfy alternatives (i)–(iii) of eligibility. This occurs in the following scenario: A blossom step makes e a blossom edge of B with (ii) holding. Next an augmenting path passes through B, changing e from matched to unmatched. A subsequent search makes B an inner blossom. Then an *expand* step is done for B, adding e to $\mathcal{S}$ (as in Figure 5). Now e is an unmatched edges in $\mathcal{S}$,

not in a current blossom, with $yz(e) = c(e) - 2$. Thus e does not satisfy any of alternatives (i)–(iii).

To remedy this, phases 2–3 use the weaker criteria for eligibility. This makes the above edge e eligible. The weaker criteria suffice for these phases. (They would not be adequate for phase 1, however; see Lemma 5.1.)

This concludes the description of *path*. Showing that *path* is correct amounts to checking that it accomplishes the goal stated in the *match* routine: It dissolves all old blossoms on $P(R)$ while maintaining (1)–(2). The discussion above shows that (1)–(2) are preserved. (Note that any edge e in $\mathcal{S}$ has $yz(e) \geq c(e) - 2$; hence, e satisfies (1b) if it gets matched or enters a blossom.) The blossoms on $P(R)$ may all dissolve in phase 1 or 2. Otherwise they dissolve in phase 3 because of the halting condition. When $R = G^*$, since G is critical the entire graph eventually becomes a blossom. This dissolves the old blossom G^*, and *path* halts correctly.

This argument applies to the first scale, even though T_0 need not be a blossom tree. In the first scale *path*(R) is trivial except when $R = G^*$. In this case *path* alternates between Augment Steps and *shell_search*(G^*), and works as desired. Note also that the first scale can detect noncritical input graphs if desired: G is critical if and only if the first scale halts with a blossom G^*.

5. *Efficiency*: *High-Level Analysis*

This section analyzes the running time of *path*. It assumes an inequality that is derived in Section 6, thus completing the high-level analysis.

The efficiency of the algorithm depends on the fact that all quantities are integral. Let us verify this fact. First, observe that an execution of *shell_search* keeps all values integral if two conditions are met: (i) all costs are even; (ii) the search starts with all y values of free vertices having the same parity. The proof that these two conditions suffice is essentially the same as the proof of the integrality property of Edmonds' algorithm, sketched in Section 1.1. It is modified to account for the fact that the search graph $\mathcal{S}$ consists of eligible edges rather than tight edges. This amounts to the observation that $c(e) - 2$ is even if $c(e)$ is.

Thus to prove integrality it suffices to show that the scaling algorithm maintains conditions (i)–(ii). Condition (i) holds because the Double Step makes all costs even. For (ii), we show that throughout the scaling algorithm, any free vertex v has $y(v)$ odd. A Double Step makes $y(v)$ odd. Suppose a call *shell_search*(C, D) does a dual adjustment by δ. If $v \in G(C, D)$, then $y(v)$ does not change—the dual adjustment increases it by δ (since v is free) and the corresponding translation of C decreases it by δ. If $v \in D$, then the translations decrease $y(v)$ by 2δ, so it remains odd. These are the only changes to $y(v)$.

In summary, we have shown that all quantities in the scaling algorithm are integral. For instance, each dual adjustment quantity δ is integral.

Now we present the properties that limit the number of iterations in the three phases. Clearly there is at most one phase 3 iteration. Phases 1 and 2 make progress by either adjusting the duals or augmenting the matching.

A phase 2 iteration need not adjust duals. This is because of the definition of eligibility. However, any phase 2 iteration after the first augments the

matching. To see this, observe that in phase 2 *shell_search* halts with an even shell only if an augmenting path has been found. The same is true if it halts with an odd shell that contains all free vertices of R', when there is more than one free vertex. So if a phase 2 Search Step halts without finding an augmenting path, either all blossoms are dissolved or there is precisely one shell, an odd shell with one free vertex. In the first case *path* terminates and in the second it starts phase 3.

A phase 1 iteration may not augment the matching, as remarked above. We now show that any phase 1 iteration does a dual adjustment. Note that in bipartite matching the analogous statement is true because there are no augmenting paths of eligible edges when a search starts [16]. This is not true for general graphs—an augmenting path of eligible edges may exist when *shell_search* is called. We show that the duals get adjusted nonetheless.

LEMMA 5.1. *In phase* 1 *after the first Search Step, any execution of shell_search*(S) *adjusts duals by* $\delta = 1$.

PROOF. By integrality it suffices to show that *shell_search* adjusts duals by some positive amount. Suppose for the sake of contradiction that *shell_search*(S) halts before doing a dual adjustment, because it finds an augmenting path of eligible edges P. Let $\mathscr{A}$ denote the preceding Augment Step and let H denote the auxiliary graph in $\mathscr{A}$.

First observe that P corresponds to a path $\bar{P}$ in H. This follows if we show that the edges of P were eligible at the start of $\mathscr{A}$ (for then each edge of P is either in H or in a blossom contracted in H). An Augment Step does not create an eligible edge. Also *shell_search*(S) can do *grow* and *blossom* steps, but no *expand* steps (as observed above, at the start of *shell_search*(S) every current root blossom is weighted). It follows that *shell_search*(S) does not create an eligible edge. This gives the desired conclusion.

Next observe that $\bar{P}$ is a simple path in H. This follows because if B is a blossom contracted in H, *shell_search*(S) did not expand B, so $E(P) \cap E(G(B))$ is a subpath (possibly empty) of P.

We conclude that $\bar{P}$ is an augmenting path in H at the end of $\mathscr{A}$. Now the maximality criterion in the Augment Step implies that in H, $\bar{P}$ contains a vertex v of some path $Q \in \mathscr{P}$. Thus $\bar{P}$ contains the matched edge e incident to v. But e became ineligible in the augment of Q. (This depends on the definition of eligibility in phase 1.) This is the desired contradiction. □

We can now do the high-level timing analysis for *path*(R), where R is any major path root. For convenience, write $\hat{n} = \hat{n}(R)$ and $\hat{m} = \hat{m}(R)$. Recall that ρ is the number of iterations of the loop of *path* in phase 1. Let R_1 be the largest undissolved blossom in $P(R)$ at the end of phase 1. Let F_1 denote the set of free vertices of R_1. The number of phase 2 iterations is at most $1 + (|F_1|/2)$ since every phase 2 iteration after the first augments the matching. Assume for the moment that this *product inequality* holds:

$$(\rho - \log \hat{n})(|F_1| - 1) \leq 5\hat{n} \log \hat{n}.$$

Thus, if $\rho \geq 2 \log \hat{n}$, then the number of phase 2 iterations is at most $(5\hat{n} \log \hat{n})/\rho + 3/2$.

Recall the time bounds for the various phases, as already mentioned and presented in detail in Section 7: $O(\hat{m})$ for one iteration in phase 1,

$O(\hat{m}\alpha(m, n))$ for one iteration in phase 2 plus $O(\hat{n} \log \hat{n})$ total extra time, and $O(\hat{m} \log \hat{n})$ total time for phase 3. Then the total time for *path*(R) is $O(\rho\hat{m} + ((\hat{n} \log \hat{n})/\rho)\hat{m}\alpha(m, n) + \hat{m} \log \hat{n})$. If $\sqrt{\hat{n}} \geq 2 \log \hat{n}$, then take

$$\rho = \sqrt{\hat{n}\alpha(m, n)\log \hat{n}}\ .$$

This gives time $O(\sqrt{\hat{n}\alpha(m, n)\log \hat{n}}\ \hat{m})$ for *path*(R). If $\sqrt{\hat{n}} < 2 \log \hat{n}$ then the time is $O(1)$ so the previous bound holds again. Now Lemma 3.1 gives the desired time bound for the entire algorithm.

To complete the timing analysis, we need only prove the above product inequality. We now show this inequality follows from the "witness inequality" defined below. To state the witness inequality, we first introduce two quantities that are fundamental in the next section. For a vertex v and an old blossom B, at any time in *path* define

$\hat{\delta}(B)$ = the total of all translations of B;

$\hat{\delta}(v, B)$ = the total of all translations of B made when v is in an undissolved shell with outer boundary B.

Since translating B by one decreases $z(B)$ by two, $\hat{\delta}(B) \leq z_0(B)/2$; equality holds if B dissolves before becoming a current blossom. The quantity $\hat{\delta}(v, B)$ counts all translations of B "witnessed by" v. It is positive only if $v \in B$. There can be more than one inner boundary of shells contributing to $\hat{\delta}(v, B)$. We often write expressions like $\hat{\delta}(F, P)$, where F is a set of vertices and P a set of blossoms (for instance the blossoms in a path of the blossom tree); recall that by our conventions for functions this denotes $\sum\{\hat{\delta}(v, B) \mid v \in F,\ B \in P\}$.

Now choose any time in *path*(R). Let ω be a free vertex in the innermost possible blossom of $P(R)$. Let F denote the set of free vertices of $P(R)$. The *witness inequality* is

$$\hat{\delta}(F - \omega, P(R)) \leq 5\hat{n} \log \hat{n}.$$

(This inequality is one reason why Section 6 analyzes the algorithm in terms of $\hat{\delta}$ rather than the quantities y, z directly involved in the algorithm. Intuitively, $\hat{\delta}(v, \cdot)$ is directly related to progress made by the algorithm, since the translations witnessed by a free vertex v correspond to searches for an augmenting path involving v. On the other hand $y(v)$ does not change when progress is being made by searching for an augmenting path from v — recall Section 4, modification (ii) of *shell_search*.)

To derive the product inequality, consider any vertex $v \in F_1$. In the Sort Step of any iteration of *path*, let $S(v)$ denote the undissolved shell of $P(R)$ that contains v. We show that the Search Step adjusts the duals in $S(v)$ in all but at most $\log \hat{n}$ phase 1 iterations of *path*. All but possibly one phase 1 execution of *shell_search*$(S(v))$ adjusts duals by $\delta = 1$ (Lemma 5.1). The Search Step does not execute *shell_search*$(S(v))$ only if a boundary of $S(v)$ dissolves before $S(v)$ is examined. In this case the ordering of the Sort Step implies that the quantity $\hat{n}(S(v))$ doubles. This can happen only $\log(\hat{n}/2)$ times, since a shell has at least two vertices. Thus, the Search Step adjusts duals in $S(v)$ in all but at most $1 + \log(\hat{n}/2) = \log \hat{n}$ iterations, as desired.

We conclude that in the ρ iterations of phase 1, v witnesses at least $\rho - \log \hat{n}$ translations, that is, $\hat{\delta}(v, \mathrm{P(R)}) \geq \rho - \log \hat{n}$, and $\hat{\delta}(F - \omega, P(R))$

$\geq (\rho - \log \hat{n})(|F_1| - 1)$. This together with the witness inequality obviously implies the product inequality.

6. *The Witness Inequality*

This section derives the witness inequality, thereby completing the high-level efficiency analysis for *path*. The section ends by proving a related inequality needed for the implementation of *shell_search*.

We start with terminology. The derivation centers around the old blossom tree T_0. We use an interval notation for paths in T_0: If node C is an ancestor of D, $[C, D]$ denotes the path from C to D with both endpoints included; $[C, D)$ is the same path with D excluded, etc. For an odd shell $G(C, D)$ of $P(R)$, any interval ending with D, e.g., $[C, D)$, is interpreted as if D were the last node of $P(R)$ (notice that $D = \emptyset$ for an odd shell). Recall that G^* is the root of the old blossom tree T_0, so $[G^*, C]$ is the path from the root to C.

If B is a node of a tree, $\mathcal{N}(B)$ denotes the set of its nonmajor children and $\mathcal{D}(B)$ denotes the set of its descendants (including B). These functions can also be applied to sets of nodes, e.g., if P is a path in a tree, $\mathcal{D}\mathcal{N}(P)$ denotes the set of all descendants of nonmajor children of nodes of P. If P is an interval, we omit the enclosing parentheses in these notations, so $\mathcal{N}[C, D]$, $\mathcal{N}[C, D)$, etc. have the obvious meanings.

Edge e *crosses* a set of vertices B if precisely one of its ends is in B, that is, $|e \cap B| = 1$. (In this notation B is usually a blossom.) The *crossing function* $\gamma: 2^{V(G)} \to \mathbf{Z}$ of a matching M is defined by $\gamma(B) = |\{e \mid e \in M \text{ crosses } B\}|$. For example if B is a blossom of M then $\gamma(B) \leq 1$. A blossom B is *uncrossed* if $\gamma(B) = 0$; otherwise it is *crossed*. A shell $G(C, D)$ (even or odd) is *uncrossed* if all its boundaries are uncrossed; otherwise it is *crossed*. During the algorithm an old blossom is uncrossed if it is undissolved; it is crossed if all its vertices have become matched; from when it dissolves until all its vertices become matched, it can alternate arbitrarily between crossed and uncrossed.

The first step in the derivation is to summarize the changes in duals y, z caused by scaling and *shell_searches*. This leads to an inequality similar to the witness inequality, Lemma 6.1. To state it, fix a time in the execution of *path*(R). Let M be the current matching. Let γ be the crossing function for M. Choose a free vertex ω in the innermost possible blossom of $P(R)$. Let M_0 be the ω-matching on R given by the 2-optimum matching of the previous scale. Let γ_0 be the crossing function for M_0. Thus an old blossom B has $\gamma_0(B) =$ **if** $\omega \in B$ **then** 0 **else** 1. (In the first scale $R = G^*$, ω can be any vertex and M_0 any ω-matching.)

Let $G(C, D)$ be an uncrossed shell of $P(R)$. (Bear in mind that C or D may be currently dissolved or undissolved, and D can be $\emptyset$ for an odd shell.) Let F_ω be the set of free vertices of $G(C, D) - \omega$. (Possibly $F_\omega = \emptyset$.) Recall that the set of old blossoms is $V(T_0)$ and the set of current blossoms is $V(T)$. In the following lemma all time-dependent quantities $(\gamma, \hat{\delta}, F_\omega)$ are evaluated at the chosen time in the execution of *path*(R).

LEMMA 6.1. *At any time in path*(R) *an uncrossed shell* $G(C, D)$ *of* $P(R)$ *satisfies*

$$(\gamma - \gamma_0)\hat{\delta}\big((C, D) \cup \mathcal{D}\mathcal{N}[C, D)\big) + \hat{\delta}\big(F_\omega, V(T_0)\big) \leq 5\hat{n}(C, D).$$

PROOF. We start with some terminology. We frequently use our convention of identifying a subgraph with its vertices or edges, e.g., $M \cap G(C, D)$ abbreviates $M \cap E(G(C, D))$. Define

$$M' = M \cap G(C, D), \qquad M_0' = M_0 \cap G(C, D),$$
$$d = c(M_0') - c(M'), \qquad \mu(B) = |M_0' \cap G(B)| - |M' \cap G(B)|.$$

In the last definition B is a blossom, old or current. Say that an old blossom and a shell "intersect" if they have a common vertex. Thus, the old blossoms that intersect $G(C, D)$ are those in $[G^*, D) \cup \mathscr{DN}[C, D)$. The argument is based on estimating d in two ways.

Observe that neither M nor M_0 crosses C or D. For M, this holds by hypothesis. Since M does not cross C and $\hat{n}(C)$ is odd, $\omega \in C$. (This depends on the fact that $G(C, D)$ is a shell of $P(R)$.) Hence, M_0 does not cross C. Similarly, M_0 does not cross D if $D \neq \varnothing$.

First estimate d using the initial duals y_0, z_0. Conditions (1) of the previous scale and the Double Step of the *scaling routine* imply

$$y_0 z_0(e) \leq c(e), \qquad \text{for} \quad e \in M;$$
$$y_0 z_0(e) \geq c(e) - 8, \qquad \text{for} \quad e \in M_0.$$

(This holds for the first scale $s = 1$, since $y_0 z_0(e) = -2$ and $|c(e)| \leq 2$.) Adding the M inequalities and subtracting the M_0 inequalities for the edges of $M' \cup M_0'$ gives

$$-y_0(F_\omega) + \mu z_0(V(T_0)) \leq -d + 8|M_0'|.$$

This inequality depends on the fact that neither matching crosses C or D. Recall that by the conventions of Section 1, $\mu z_0(V(T_0))$ denotes $\sum\{\mu(B) z_0(B) \mid B \in V(T_0)\}$.

Next estimate d in a similar way using the current duals y, z. Since (1) holds for the current duals, adding (1a) for M_0' and subtracting (1b) for M' gives

$$y(F_\omega) - \mu z(V(T_0) \cup V(T)) \leq d + 2|M'|.$$

This also depends on the fact that neither matching crosses C or D.

Next we bound the terms involving $V(T)$ and $V(T_0)$ in the two d estimates. A current blossom $B \in V(T)$ has $\mu(B) \leq 0$. This follows since $|M' \cap G(B)|$ is as large as possible; this in turn follows since M induces a u-matching on B for some $u \in B$, and no edge of M crosses C or D. (Note however that an edge of B can cross C or D.) Since μ is nonpositive on $V(T)$, the $V(T)$ term in the second d estimate can be dropped.

We turn to the $V(T_0)$ terms. First note that an argument similar to the above shows that μ is nonnegative on old blossoms. This fact will be used below.

Clearly, μ vanishes on blossoms not intersecting $G(C, D)$, so we can restrict attention to old blossoms B intersecting $G(C, D)$. Define

$$f(B) = |F_\omega \cap V(B)|;$$

in addition, define γ' and γ_0' as the crossing functions of M' and M_0', respectively. We show the following inequality in order to bound the $V(T_0)$ terms:

$$\mu(z_0 - z)(B) \geq (f + \gamma' - \gamma_0')\hat{\delta}(B).$$

To prove this, we first prove the equation $2\mu(B) = (f + \gamma' - \gamma_0')(B)$: By definition $2\mu(B)$ is how many more vertices of $V(B) \cap G(C, D)$ that M_0 matches on edges of $G(B) \cap G(C, D)$ compared to M. A vertex of $V(B) \cap G(C, D)$ is not matched on an edge of $G(B) \cap G(C, D)$ if it is free or it is matched on an edge crossing $V(B) \cap G(C, D)$. There are $f(B)$ vertices of the first type (note that ω is free in both matchings). There are $(\gamma' - \gamma_0')(B)$ vertices of the second type, since no edge of either matching crosses C or D. This gives the desired equation.

Now we show that an old blossom B has $(z_0 - z)(B) \geq 2\hat{\delta}(B)$ or $\mu(B) = 0$. Note that this relation, together with the above equation and the nonnegativity of $\mu(B)$, implies the desired inequality for the $V(T_0)$ terms. The relation follows by considering three cases. If B has never become a current blossom then $(z_0 - z)(B) = 2\hat{\delta}(B)$. If B has become a current blossom, but is not now a current blossom, then $(z_0 - z)(B) = z_0(B) \geq 2\hat{\delta}(B)$. If B is a current blossom, then $\mu(B) = 0$. The desired relation follows, as does the inequality for $V(T_0)$ terms.

Next we deduce

$$(y - y_0)(F_\omega) + (f + \gamma' - \gamma_0')\hat{\delta}\big([G^*, D) \cup \mathscr{M}[C, D)\big) \leq 5\hat{n}(C, D).$$

This follows by adding the two d estimates and replacing the $\mu(z_0 - z)$ terms using the above observations. In addition, note that $|M'| \leq |M_0'| \leq \hat{n}(C, D)/2$.

A free vertex v has $y(v) = y_0(v) - (\hat{\delta} - \hat{\delta}(v, \cdot))[G^*, v)$. (To show this, observe that if v is in a blossom that gets translated by δ, then $y(v)$ decreases by δ; if v is in a shell that gets duals adjusted by δ, then $y(v)$ increases by δ; these are the only times that $y(v)$ changes.) Summing these equations for all $v \in F_\omega$ implies

$$(y - y_0)(F_\omega) + f\hat{\delta}\big([G^*, D) \cup \mathscr{M}[C, D)\big) - \hat{\delta}\big(F_\omega, V(T_0)\big) = 0.$$

Finally, subtract the last equation from the preceding inequality. This gives the lemma if we use two observations to simplify the expression $(\gamma' - \gamma_0')\hat{\delta}([G^*, D) \cup \mathscr{M}[C, D))$: The terms $(\gamma' - \gamma_0')\hat{\delta}([G^*, C])$ vanish, since a blossom $B \in [G^*, C]$ has $\gamma'(B) = \gamma_0'(B) = 0$. The remaining terms are for blossoms B contained in $G(C, D)$. Hence the function $\gamma' - \gamma_0'$ simplifies to $\gamma - \gamma_0$. □

Rewrite the lemma as

$$\begin{aligned}(\gamma - \gamma_0)\hat{\delta}\big((C, D)\big) + \hat{\delta}\big(F_\omega, P(R)\big) \\ \leq 5\hat{n}(C, D) + \big((\gamma_0 - \gamma)\hat{\delta} - \hat{\delta}(F_\omega, \cdot)\big)\mathscr{M}[C, D)\end{aligned} \tag{3}$$

Let us interpret this inequality and survey the rest of the derivation. Each scale after the first starts off with an "error" of $O(n)$, in the sense that the 2-optimum matching of the previous scale can cost $O(n)$ more than that of the current scale. If the graph is bipartite, this is the only source of error [16]. For general graphs there is a second type of error when *path*(R) begins. It comes from changes in the duals made by previous calls to *path* for descendants of R. Specifically, the error from scaling corresponds to the first term on the right-hand side of (3) and the error from descendants corresponds to the second

term. Loosely speaking the second term is bounded by the total amount of all dual adjustments in phases 1–2. We will show this in a precise sense, in the process of proving that the second term is $O(\hat{n}(C, D)\log \hat{n}(R))$. Then (3) will imply that $\hat{\delta}(F_\omega, P(R))$ is $O(\hat{n}(R)\log \hat{n}(R))$, the desired witness inequality.

We begin with a convenient notation that corresponds to "total dual adjustment." First let us clarify a phrase used throughout this section, *a search of shell* $G(A, B)$. This refers to the operation of *shell_search* when it is working on shell $G(A, B)$. Notice that it is possible for this search to occur during an execution of *shell_search*(S) where $S \neq G(A, B)$: If S is a proper subshell of $G(A, B)$ then after one or more *dissolve* steps the search can be working on $G(A, B)$.

For an even shell $G(C, D)$ of $P(R)$ define

$$\Delta(C, D) = \text{the total of all translations in searches of shells } G(A, B) \text{ for } A, B \in [C, D].$$

Equivalently, $\Delta(C, D)$ is twice the total of all dual adjustments made in searches of shells $G(A, B)$, for $A, B \in [C, D]$; Δ is evaluated after the last translation of a shell in $G(C, D)$. (Note that $\Delta(C, D)$ differs slightly from "the total of all translations in executions *shell_search*(A, B) for $A, B \in [C, D]$." If an execution *shell_search*(C, D) dissolves C or D and proceeds to adjust duals by some positive amount, $\Delta(C, D)$ does not count the corresponding translations whereas the alternative definition would.)

The main step in the derivation discussed above is to show that any even shell $G(C, D)$ of $P(R)$ has

$$\Delta(C, D) \leq 5\hat{n}(C, D)\lfloor \log \hat{n}(R) \rfloor. \tag{4}$$

We prove (4) by induction on $\hat{n}(R)$. The base case, R a leaf blossom, is vacuous. The inductive step is done in Lemmas 6.2–6.3 and the following paragraph. The lemmas inductively assume (4) for shells in major paths of descendants of R.

We start by bounding the error from descendants. (This essentially gives the witness inequality.) In the following lemma, all notation is as in Lemma 6.1.

LEMMA 6.2. *At any time in path*(R) *an uncrossed shell* $G(C, D)$ *of* $P(R)$ *satisfies*

$$(\gamma - \gamma_0)\hat{\delta}((C, D)) + \hat{\delta}(F_\omega, P(R)) \leq 5\hat{n}(C, D)\lfloor \log \hat{n}(R) \rfloor.$$

PROOF. By (3) we need only show

$$((\gamma_0 - \gamma)\hat{\delta} - \hat{\delta}(F_\omega, \cdot))\mathscr{D}\mathscr{N}[C, D) \leq 5\hat{n}(C, D)(\lfloor \log \hat{n}(R) \rfloor - 1). \tag{5}$$

The left-hand side of (5) is made up of contributions coming from translations made prior to the call *path*(R). More precisely consider a shell $G(A, B)$ of $P(S)$, where S is a major path root descending from $\mathscr{N}[C, D)$. Let $f = |F_\omega \cap G(A, B)|$. Suppose that a search of $G(A, B)$ does a dual adjustment of δ. The corresponding translations contribute $((\gamma_0 - \gamma)(\{A, B\}) - f)\delta$ to the left-hand side. The first step is to show that this contribution is positive only for uncrossed even shells containing no free vertices.

The number of vertices of $G(A, B) - \omega$ not matched on edges of $G(A, B)$ has the same parity in the old matching M_0 as in the current matching M.

(Recall that ω is free in both matchings; also note that $G(A, B)$ can be odd or even.) This number is $\gamma_0(\{A, B\})$ for M_0, and $\gamma(\{A, B\}) + f$ for M. Thus $(\gamma_0 - \gamma)(\{A, B\}) - f$ is even. Since $\gamma_0(\{A, B\}) \leq 2$, the contribution is positive only if $\gamma_0(\{A, B\}) = 2$ and $\gamma(A) = \gamma(B) = f = 0$, that is, $G(A, B)$ is an uncrossed even shell with no free vertices, as claimed.

Let $G(A, B)$ be an uncrossed even shell with no free vertices. Any blossom descending from $\mathcal{N}[A, B)$ is crossed (since it has no free vertices). Thus, any uncrossed even shell with no free vertices that intersects $G(A, B)$ is a shell of $P(S)$. Among these shells is a maximal one. Let $G(A, B)$ denote such a maximal shell.

The maximal shells $G(A, B)$ are vertex disjoint. Ignoring the negative terms on the left-hand side of (5) shows that $G(A, B)$ contributes at most $\Delta(A, B)$. The inductive assumption (4) implies $\Delta(A, B) \leq 5\hat{n}(A, B)\lfloor \log \hat{n}(S) \rfloor$, where S is a descendant of a nonmajor child of $P(R)$, that is, $\hat{n}(S) < \hat{n}(R)/2$. Thus $\Delta(A, B) \leq 5\hat{n}(A, B)(\lfloor \log \hat{n}(R) \rfloor - 1)$. Summing over all the disjoint shells $G(A, B)$ gives (5). □

Now we estimate Δ for even shells. Call $G(C, D)$ *active* if during the course of the algorithm there is a search of shell $G(C, D)$ that adjusts duals by some positive amount. For an active shell $G(C, D)$ consider the last dual adjustment. Let M be the matching at the time of this last dual adjustment and let γ be the crossing function of M. Note there is a natural correspondence between intervals $[A, B)$ in the blossom tree and shells $G(A, B)$. Call an interval *even*, *uncrossed*, *active*, etc. if the corresponding shell is.

LEMMA 6.3. *An active interval* $[C, D)$ *of* $P(R)$ *can be partitioned into uncrossed intervals* $[C, C')$, $[C', D')$ *and* $[D', D)$ *such that if* F *is the set of free vertices of* M *in* $[C', D')$,

$$\Delta(C, D) \leq \gamma\,\hat{\delta}((C', D')) + \hat{\delta}(F, [C, D)) + \Delta(C, C') + \Delta(D', D).$$

PROOF. Choose any free vertex v in $G(C, D)$ and let $G(C', D')$ be the minimal uncrossed shell containing v. (Thus C' is the innermost blossom of $P(R)$ containing v with $\gamma(C') = 0$ and D' is the outermost blossom of $P(R)$ not containing v with $\gamma(D') = 0$.) Clearly $[C', D') \subseteq [C, D)$ since neither C nor D is crossed at the time of the last dual adjustment. Possibly $C = C'$ or $D = D'$.

Consider any shell $G(A, B)$ that gets searched, with $A, B \in [C, D]$. If $A, B \in [C, C']$ then the translations for this shell are counted by the term $\Delta(C, C')$ on the right-hand side; similarly if $A, B \in [D', D]$. The remaining possibility is that $G(A, B)$ intersects $G(C', D')$. Let $G(A', B')$ denote the intersection, that is, $A' =$ **if** $A \in (C', D')$ **then** A **else** C', $B' =$ **if** $B \in (C', D')$ **then** B **else** D'. Let f denote the number of free vertices in $G(A', B')$. Since $G(A', B')$ is an even shell, $\gamma(\{A', B'\}) + f$ is even. This quantity is positive, since if $\gamma(A') = \gamma(B') = 0$ then the minimality of $G(C', D')$ implies that $A' = C'$ and $B' = D'$; thus $f > 0$. We conclude that $\gamma(\{A', B'\}) + f \geq 2$.

In a search of $G(A, B)$, a dual adjustment of δ contributes 2δ to the left-hand side of the inequality of the lemma. To show that it contributes at least this much to the right, observe that $\gamma(\{A', B'\})\delta$ is counted by the first term on the right, $f\delta$ by the second. Furthermore $\gamma(A') > 0$ only if $A' = A$, by the

above formula for A'. Hence the term $\gamma\hat{\delta}(A')$ on the right only counts translations of A. The same holds for B', so our method of counting is valid. □

Now we complete the proof of the inductive assertion (4). First observe that the active intervals are nested, that is, if two active intervals intersect then one contains the other. Thus any even interval $[A, B)$ of $P(R)$ has a unique set of maximal active subintervals $[C, D)$ such that $\Delta(A, B)$ equals the sum of all values $\Delta(C, D)$. We prove (4) for any even interval $[C, D)$ by induction on $\hat{n}(C, D)$. If $[C, D)$ is not active, apply the inductive assertion to each maximal active subinterval to get (4) for $[C, D)$. Suppose $[C, D)$ is active. Apply Lemma 6.3. The first two terms on the right, $\gamma\,\hat{\delta}((C', D')) + \hat{\delta}(F,[C, D))$, are bounded above by $5\hat{n}(C', D')\lfloor\log\hat{n}(R)\rfloor$, by Lemma 6.2 applied to the uncrossed shell $[C', D')$. (Since D is uncrossed, $\omega \in D$ and γ_0 vanishes on (C, D).) Thus $\Delta(C, D) \le 5\hat{n}(C', D')\lfloor\log\hat{n}(R)\rfloor + \Delta(C, C') + \Delta(D', D)$. The desired conclusion follows from the inductive hypothesis for $[C, C')$ and $[D', D)$. Thus we have proved (4), and in the process shown that Lemmas 6.2–6.3 hold with no assumptions.

Now we derive the witness inequality. Let R be a major path root. At any time in the execution of *path*(R) let F be the set of free vertices of R, and let ω be a free vertex in the innermost possible blossom of R. The choice of time in *path*(R) implies that $\gamma(R) = 0$ and ω exists. Note that a blossom $B \in P(R)$ has $\gamma(B) \ge \gamma_0(B)$ (since $\gamma_0(B) =$ **if** $\omega \in B$ **then** 0 **else** 1, and $\omega \notin B$ implies $\gamma(B) > 0$). Thus Lemma 6.2 applied to the uncrossed shell R gives the witness inequality,

$$\hat{\delta}\big(F - \omega, P(R)\big) \le 5\hat{n}(R)\log\hat{n}(R).$$

We conclude with a related inequality used to implement the priority queue for *shell_search* (in Section 7).

COROLLARY 6.1. *For a major path root R, the total dual adjustment in all phase 2 shell_searches of path(R) is at most $5\hat{n}(R)\log\hat{n}(R)$.*

PROOF. Write the total dual adjustment in phase 2 as $d_1 + d_2$, where d_1 is the total adjustment when there are free vertices in R' that are not in the (undissolved) odd shell, and d_2 is the remainder, that is, the total adjustment when all free vertices in R' are in the odd shell. We show that each $d_i \le (5/2)\hat{n}(R)\log\hat{n}(R)$.

The dual adjustments counted in d_1 all occur in a search of an even shell. If D is the smallest blossom in $P(R)$, (4) shows that $\Delta(R, D) \le 5\hat{n}(R, D)\lfloor\log\hat{n}(R)\rfloor$. Thus the definition of Δ implies that $d_1 \le (5/2)\hat{n}(R)\log\hat{n}(R)$.

For d_2, consider the odd shell $G(R', \varnothing)$ immediately after the last dual adjustment counted in d_2. The definition of phase 2 implies that at this time the odd shell has at least three free vertices v. Each such vertex has witnessed every dual adjustment of an odd shell in phase 2, that is, $\hat{\delta}(v, P(R)) \ge d_2$. Thus the witness inequality implies that $d_2 \le (5/2)\hat{n}(R)\log\hat{n}(R)$. □

7. *The Search Step*

This section gives the data structures and details of the Sort and Search Steps.

We begin with a data structure needed for the Search Step. At the start of the *match* routine, the old blossom tree T_0 is ordered so that every major child is a

rightmost child. The vertices of G, which are the leaves of T_0, are numbered from 1 to n in left-to-right order. In the following discussion we identify each vertex with its number. Each node B of T_0 stores $lo(B)$, its lowest-numbered leaf descendant. The given graph G is represented by adjacency lists, two lists for each vertex v. One list for v contains the edges $\{vw \mid w < v\}$ ordered by decreasing w. The other list contains the edges $\{vw \mid w > v\}$ ordered by increasing w.

This data structure is constructed (once in each scale) in time $O(m)$ using a bucket sort. The main property of the vertex order is that in any execution of $path(R)$, the vertices of an undissolved shell $G(C, D)$ (even or odd) constitute the interval $[lo(C) \,..\, lo(D))$. Hence for any vertex v in an undissolved shell $G(C, D)$, the edges incident to v in $G(C, D)$ can be found by scanning the appropriate part of v's two adjacency lists (assuming the values $lo(C)$, $lo(D)$ are known). The time for this scan is $O(1)$ plus time proportional to the number of edges found in $G(C, D)$.

Now consider an execution of $path(R)$. As in Section 5 let $\hat{n} = \hat{n}(R)$ and $\hat{m} = \hat{m}(R)$. The undissolved blossoms of $P(R)$ are stored in a doubly-linked list $\mathscr{U}$; the order of blossoms in $\mathscr{U}$ is the same as in $P(R)$.

The Sort Step can be done in $O(\hat{n})$ time using a bucket sort.

We turn to the Search Step. During the course of a Search Step a vertex goes through three states: it starts out *asleep*; while it is in the shell being searched it is *active*; when its shell has been searched it is *dead*. (Vertices outside of the largest undissolved shell R' are always dead.) In greater detail, when a boundary of a shell $G(C, D)$ dissolves, the new larger shell is composed of the vertices of $G(C, D)$, currently active, and new vertices, which were either asleep or dead. If they were asleep all vertices of the new shell are active; if they were dead all vertices of the new shell are dead. (If C and D dissolve simultaneously, with one boundary introducing asleep vertices and the other dead vertices, the new shell is dead.) Thus a vertex is active in the search for only one augmenting path—once dead, it remains dead for the rest of the Search Step. All this follows from the statement of the Search Step and the halting criterion.

Consider the time in the Search Step for dissolving shell boundaries. (This is important in phases 2 and 3.) Suppose shell $G(C, D)$ is being searched and C dissolves (D dissolving is similar). Let B be the blossom preceding C in list $\mathscr{U}$, that is, the smallest undissolved blossom containing C. C is deleted from $\mathscr{U}$. Suppose the new vertices (those in $B - C$) become active. The edges in the new shell $G(B, D)$ are found by scanning the adjacency lists of the new vertices (the interval for the new shell is $[lo(B) \,..\, lo(D))$). Since a vertex becomes active only once, the total time for scanning edges in dissolve steps in one Search Step is $O(\hat{m})$. (Some additional processing that is done when a blossom dissolves is discussed below; it concerns dual values.)

Next consider the time spent in the Search Step manipulating the priority queue and doing related processing to find the next dual adjustment quantity δ (as described in Section 1.1). We consider phases 1, 3, and 2 in that order.

In phase 1 the priority queue is not needed, since only one dual adjustment with $\delta = 1$ is made. Blossom steps are implemented in linear time using the incremental tree set-merging algorithm of [15]. This makes the time for one Search Step in phase 1 $O(\hat{m})$.

The total time for phase 3 is $O(\hat{m} \log \hat{n})$. This can be achieved by imple-

menting the priority queue as a balanced tree [17]. Gabow, et al. [13] give an even better bound but this is not needed here.

For phase 2, Corollary 6.1 shows that the total dual adjustment is at most $5\hat{n} \log \hat{n}$. The priority queue is implemented using two arrays, $P[0 \,..\, \hat{n} - 1]$ and $Q[0 \,..\, 5 \log \hat{n}]$. Each array entry points to a list of priority queue entries. At any time, an edge e that will become eligible after the *shell_search* has done a total dual adjustment of d units is placed in the list for $Q(\lfloor d/\hat{n} \rfloor)$. When the total dual adjustment is in the range $[i\hat{n} \,..\, (i+1)\hat{n})$, the edges in list $Q(i)$ are stored in the lists for P, the above edge e being placed in the list for $P(d \bmod \hat{n})$. The edges in the list for $P(j)$ are eligible when the total dual adjustment reaches $i\hat{n} + j$. After a shell is searched, any entries remaining in this priority queue are removed, in linear time. This makes the total overhead for the queue in all phase 2 searches $O(\hat{n} \log \hat{n})$. (See [16] for a more detailed discussion of the implementation of such a queue.)

To implement *expand* steps the list-splitting algorithm of [10] is used. This makes the time for one Search Step in phase 2 $O(\hat{m}\alpha(m, n))$ (since a vertex is active in only one search).

The last aspect of the Search Step discussed here is maintaining the duals y, z. Most details are the same as in an efficient implementation of Edmonds' algorithm (see [13] and [17]; although the main concern of these papers is implementing the priority queue discussed above, the details needed here are also given). The main technique is using offsets to facilitate the adjustment of dual values. In addition we use offsets in connection with old blossoms and their translations. We show how the algorithm translates a blossom in $O(1)$ time, and also how it calculates $yz(e)$ in $O(1)$ time. This plus the details in [13] and [17] give the desired time bound for our algorithm.

We start by describing the data structure. The algorithm stores two values for each old blossom B, $z_1(B)$ and $t(B)$, plus a value $y'(v)$ for each vertex v. Intuitively z_1 keeps track of old z values, y' keeps track of y values and t keeps track of translations. More precisely the algorithm maintains these invariants:

$$\begin{aligned} y(v) &= y'(v) - t([G^*, v]); \\ z([G^*, B]) &= z_1(B) - 2t([G^*, B]), && \text{for } B \text{ undissolved;} \\ t(B) &= 0, && \text{for } B \text{ dissolved.} \end{aligned}$$

Initially each old blossom B has $z_1(B) = z_0([G^*, B])$ and $t(B) = 0$, and each vertex v has $y'(v) = y(v)$, so the invariant holds.

Consider a search of shell $G(C, D)$. To calculate $yz(e)$, write $yz(e) = y(e) - z([G^*, C]) - z(\{B \mid B \text{ is a current blossom containing } e\})$. The last term is calculated as in Edmonds' algorithm, so we concentrate on $y(e) - z([G^*, C])$. This equals $y'(e) - z_1(C)$, since C is the smallest undissolved blossom containing either end of e. Hence $yz(e)$ can be calculated in $O(1)$ time as claimed.

Next suppose the *shell_search* does a dual adjustment of δ. This necessitates translating blossoms C and D by δ. The algorithm does this by increasing $t(C)$ and $t(D)$ by δ. This has the same effect as a translation, maintaining the above two invariants. Hence the dual adjustment is done correctly in $O(1)$ time.

It remains only to discuss how dual values are processed when a boundary C or D dissolves in the *shell_search*. In general consider consecutive undissolved blossoms $Z \subseteq Y \subseteq X$ of $P(R)$ and suppose blossom Y has dissolved.

The invariants can be preserved by working on $G(X, Y)$ using time $O(\hat{n}(X, Y))$ or working on $G(Y, Z)$ using time $O(\hat{n}(Y, Z))$. For the first, assign $y'(v) \leftarrow y'(v) + t(Y)$ for each vertex $v \in X - Y$, then $t(X) \leftarrow t(X) + t(Y)$, $z_1(X) \leftarrow z_1(X) + 2t(Y)$ and finally $t(Y) \leftarrow 0$. For the second, assign $y'(v) \leftarrow y'(v) - t(Y)$ for each vertex $v \in Y - Z$, then $t(Z) \leftarrow t(Z) + t(Y)$ and $t(Y) \leftarrow 0$.

The algorithm adjusts duals by working on the vertices whose state changes. This implies that the time for all dissolves in a Search Step is $O(\hat{n})$.

Note that this discussion applies to both even shells and the odd shell. Observe how an odd shell $G(C, \varnothing)$ is processed: C dissolves by becoming either an unweighted or current blossom. If $C = R'$ the new odd shell is dead. In this case $t(R')$ becomes zero and, after the invariant has been restored, each vertex $v \in R$ has $y'(v) = y(v)$. In particular each scale halts with $y' = y$.

8. *The Augment Step*

This section shows that the Augment Step can be done in linear time. This amounts to solving the following problem in linear time: Given an arbitrary graph with a matching M, find a maximal set $\mathscr{P}$ of vertex-disjoint augmenting paths. We present an algorithm based on depth-first search and the properties of blossoms [6]. This section uses the following notation: F denotes the set of free vertices of M. For a vertex $v \notin F$, v' denotes the vertex matched to v.

The algorithm grows a search graph similar to the search graph $\mathscr{S}$ in Edmonds' weighted matching algorithm (Section 1.1) with three changes: First, the requirement that an edge of $\mathscr{S}$ be tight is dropped (tightness is irrelevant since there are no edge costs). Second, an inner blossom is always a vertex, never a nonleaf blossom. (This comes about because the search starts with a graph that has no blossoms. As a consequence the algorithm has no *expand* steps—only *grow* and *blossom* steps.) Third, the free vertices are added to the search graph one at a time rather than simultaneously. A free vertex f is either outer (if some search starts from f) or inner (if a search ends by finding an augmenting path to f). The contracted subgraph $\mathscr{F}$ (of Section 1.1) is always a forest: an augmenting path corresponds to a path in $\mathscr{S}$ joining a free outer vertex to a free inner vertex.

The final and most important difference from Edmonds' weighted matching algorithm is that the search is done depth-first. Figure 7 gives the recursive depth-first search procedure *find_ap*. Figure 8 shows a search graph constructed by *find_ap*. Here the input to *find_ap* is vertex 1, and vertices are labelled in the order they become outer. We explain the search algorithm by first presenting a simplified version and then discussing *find_ap*.

It is convenient to use the terminology of ordered trees. Thus the children of a vertex are ordered from left to right. Given two vertices in an ordered tree, one vertex is either an ancestor of or to the left of the other.

The goal of the simplified depth-first search is to find one augmenting path. The search maintains a search tree $\mathscr{T}$ (an ordered tree) in a graph $\overline{G}$ that is a contraction of the given graph G. It will be seen that $\overline{G}$ is the result of contracting blossoms in G. In discussing the simplified depth-first search we write G-vertex or $\overline{G}$-vertex to specify to which graph a vertex belongs. A path from a $\overline{G}$-vertex v in $\mathscr{T}$ to the root is alternating; v is *outer* (*inner*) if the length of the path is even (odd). The search maintains an *active vertex*, an outer vertex on the rightmost path of $\mathscr{T}$, from which edges are scanned.

```
procedure find_ap(x) {x is an outer vertex}
for each edge xy ∉ M do {examine an edge}
  if y ∉ V(𝒮) then
    if y is free then begin {an augmenting path has been found}
      add xy to 𝒮, and add path yP(x) to 𝒫
      terminate all active recursive calls to find_ap
      end
    else begin {two grow steps}
      add xy, yy' to 𝒮, by setting l(y*) ← x
      find_ap(y')
      end
  else if b(y) became outer strictly after b(x) then begin {blossom step}
    let u_i, i = 1, ..., k be the inner vertices in P(y, b(x)), ordered so that u_i precedes u_{i-1}
    for i ← 1 to k do begin {update 𝒮}
      l(u_i) ← (y, x)
      for each vertex v with b(v) ∈ {u_i, u_i'} do b(v) ← b(x)
      end
    for i ← 1 to k do find_ap(u_i)
    end
```

FIG. 7. Depth-first search procedure *find_ap*.

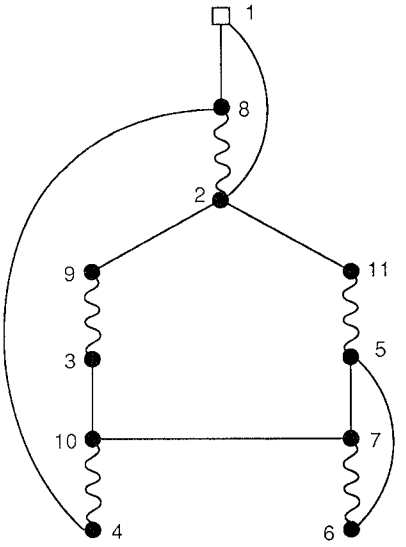

FIG. 8. Example graph for *find_ap*.

Initially $\overline{G} = G$ and $\mathcal{T}$ consists of a root which is a free vertex f of G; f is also the active vertex.

The search works by repeatedly scanning an edge incident to the active vertex. The active vertex changes as the search progresses. At any point if all edges incident to the current active vertex x have been scanned from x then the search backs up—the grandparent of x becomes the new active vertex (recall the parent of x is inner, the grandparent is outer). If the search attempts to back up from the root of $\mathcal{T}$ then it halts—it will be seen that in this case there is no augmenting path from the initially chosen free vertex f.

The procedure to scan an edge incident to the active vertex x is as follows. An edge xy that has not been scanned from x is chosen arbitrarily. First suppose $y \notin \mathcal{T}$. (It will be seen that only vertices in $\mathcal{T}$ get contracted; hence y is a G-vertex.) If y is free then the search halts (an augmenting path from f to

y has been found). Otherwise two *grow steps* are done: Recall that y' denotes the vertex matched to y. $\mathcal{T}$ is extended by adding edges xy and yy', and the active vertex becomes y' (y is made the rightmost child of x, so the new active vertex is on the rightmost path of $\mathcal{T}$). The search continues by scanning an edge incident to the new active vertex.

Now suppose $y \in \mathcal{T}$. If y is an outer descendant of x then a *blossom step* is done: The path from y to x, plus edge xy, forms a blossom. It is shrunk to a new vertex B giving a new contracted graph $\overline{G}$. The active vertex becomes B (clearly B is on the rightmost path). The search continues by scanning an edge incident to the new active vertex. (Note that when $y \in \mathcal{T}$ nothing is done if y is not an outer descendant of x, even though in some cases a new blossom could be shrunk.)

This completes the description of the simplified algorithm. It finds an augmenting path leading to the free vertex f if one exists. We shall not prove this completely (it follows from arguments similar to Lemma 8.1 below). Instead we prove two properties, designated (i) and (ii), needed to analyze *find_ap*. Property (i) is that if v and w are adjacent outer vertices and edge vw has been scanned from both vertices then v and w are in the same blossom. (Although it is not logically necessary for our development, we remark that property (i) essentially proves that the simplified algorithm works correctly when it halts without finding an augmenting path: The property implies no more blossom steps are possible, which in turn implies that no augmenting path exists.)

We first show that the algorithm maintains the invariant that no edge goes from an outer vertex to the right. It is easy to see that at any time any outer vertex not on the rightmost path has been completely scanned. Thus, in a grow step, no outer vertex to the left of the rightmost path is adjacent to y or y'. Thus grow steps preserve the invariant. A blossom step preserves the invariant, since in general contracting an edge in a tree does not move a vertex to the left or right of another. We conclude that the invariant always holds.

We make a second observation. Consider edges xy, yy' added to $\mathcal{T}$ in two grow steps that make y inner and y' outer. Suppose that after y' is made active the search eventually backs up from y'. Then if y' is ever contained in the active vertex again, edges xy and yy' are contracted. To prove this consider the first time y' or one of its descendants again enters the active vertex. This occurs in a blossom step that contracts a path from the current active vertex, a nondescendant of y', to a descendant of y'. Clearly edges xy and yy' are on this path and get contracted.

Now we prove property (i). Suppose that at any time an edge vw joins two distinct outer vertices of $\overline{G}$. The invariant implies that one vertex, say v, is an ancestor of the other, w. We show that if edge vw gets scanned from vertex v (or from some $\overline{G}$-vertex containing v) then v and w are vertices in the same blossom. (Clearly this implies (i).) Consider the time when vw is scanned from v, and assume w is not in the same blossom as v. Vertex w is in $\mathcal{T}$ when vw is scanned (otherwise a grow step would make w a child of v; but then vw never joins two distinct outer blossoms). Hence w is inner or outer at this time.

Suppose w is inner. Let a be the first ancestor of w on the path from the active vertex v to the root. Observe that the search has backed up from every outer vertex on the path P from w' to a, except a. By the time w becomes outer, all vertices on P are in the same blossom as a (this follows from the second observation above). But since a is an ancestor of v, v is never a proper

ancestor of the outer vertex w. This contradiction shows that w cannot be inner.

We conclude that w is outer when vw is scanned from v. Furthermore w is a descendant of v (since edge contractions cannot change a proper ancestor into a proper descendant). Hence a blossom step is done, placing v and w in the same blossom, as claimed. This proves (i).

Now we extend the simplified algorithm to record information about G. Call a G-vertex *outer* if it is contained in an outer $\bar{G}$-vertex. The search implicitly constructs an even-length alternating path from each outer G-vertex x to the free vertex f; let $P(x)$ denote this path. We extend the algorithm to make the paths $P(x)$ explicit, using the following data structure ([8]).

In this discussion interpret a path P as an ordered list of vertices. For example the first vertex of $P(x)$ is x. Let P^r denote the reverse path of P; if Q is also a path let PQ denote the concatenation of the two paths. (For this to be a path the last vertex of P must be adjacent to the first vertex of Q.) For vertex $y \in P(x)$, let $P(x, y)$ denote the subpath of $P(x)$ from x to y.

Each outer vertex x has a label $l(x)$ that defines path $P(x)$ as follows. A label is either a *singleton label* $l(x) = y$, where y is an outer vertex, or a *pair label* $l(x) = (y, z)$, where y and z are outer vertices and the pair is ordered. If x has a singleton label $l(x) = y$ then $P(x) = xx'P(y)$. (As a degenerate case a free vertex x has singleton label $l(x) = \emptyset$ and we define $P(x) = x$.) If x has a pair label $l(x) = (y, z)$ then necessarily $x \in P(y)$, and $P(x) = P(y, x)^r P(z)$.

The algorithm assigns labels as follows. In the grow steps, y' gets the singleton label x. In the blossom step, each inner vertex u in the path from y to x gets the pair label (y, x).

Property (ii) is that the labels define $P(x)$ as an even-length alternating path from x to f. Recall that by definition an alternating path is simple. This is important for correctness of the algorithm, since it is well-known that augmenting a matching along a nonsimple augmenting path can produce a set that is not a matching. Property (ii) is easy to prove by induction. The inductive assertion also includes the fact that if x is in blossom B of $\bar{G}$, then $P(x) \cap \mathcal{T}$ is precisely the tree path from B to the root of $\mathcal{T}$.

This completes the discussion of the simplified depth-first search, which finds one augmenting path. The final version of the algorithm finds a maximal set of augmenting paths. To do this we must be more precise about selecting the active vertex. In the final algorithm the active vertex is a G-vertex. The final algorithm maintains the following property: For the active vertex x, $P(x)$ contains any outer G-vertex that has not been completely scanned. (This is analogous to ordinary depth-first search of a directed or undirected graph, where the search path leading to the vertex currently being scanned contains all vertices that are not completely scanned [1].) In order to maintain this property the blossom step works as follows: First it makes the inner vertices on the path P from y to x outer; then it makes these vertices active in the reverse order of their occurrence in P.

The final algorithm uses one other data structure, to represent the blossom structure: For each vertex x, $b(x)$ denotes the base of the root blossom containing x.

Incorporating these changes into the simplified algorithm gives the algorithm *find_ap* stated in Figure 7. Observe that for any free vertex f, *find_ap*(f) is a correct implementation of the simplified algorithm searching from f. Here we

assume that *find_ap*(f) begins with $\mathcal{P}$ empty, each $b(v)$ initialized to v and the search graph $\mathcal{S}$ initialized to root f. (Note that $\mathcal{S}$ in *find_ap* corresponds to T in the simplified algorithm; in its final usage below, $\mathcal{S}$ will contain more than one connected component, and will be analogous to the search graph $\mathcal{S}$ of Section 1.1.) Correctness depends on two observations. First, the recursion correctly implements the notion of the active vertex in $\bar{G}$. (The recursive calls in a blossom step are consistent with the simplified algorithm.) Second, the test preceding a blossom step, that $b(y)$ became outer strictly after $b(x)$, is equivalent to the simplified algorithm's test that y is in an outer descendant of x (since if $b(y)$ is outer then so is y, and edge xy implies x and y are related in the search tree of $\bar{G}$).

Since *find_ap* implements the simplified algorithm it satisfies properties (i)–(ii). In particular property (i) states that if v and w are adjacent outer vertices and edge vw has been scanned from both vertices then $b(v) = b(w)$.

We further observe the following property (iii): When *find_ap* halts, every outer vertex not in $\mathcal{P}$ has been completely scanned. To prove this it suffices to show that every time *find_ap*(x) examines an edge every unscanned outer vertex is in $P(x) \cup \mathcal{P}$. (Say that *find_ap* "examines an edge" each time control passes to the line so labelled in Figure 7. This includes the last time, when no unscanned edges xy exist.) This invariant follows easily because of the order of activating vertices u_i in a blossom step.

The final depth-first algorithm uses a main routine *find_ap_set*. This routine initializes the search graph $\mathcal{S}$ to empty and each $b(v)$ to v. Then it examines each vertex $f \in F$ in turn. If f is not in a path of $\mathcal{P}$ when it is examined, the routine adds f to S (by assigning $l(f) \leftarrow \emptyset$) and calls the recursive procedure *find_ap*(f). Note that when *find_ap* discovers an augmenting path, it immediately terminates itself and all currently active recursive calls. Also note that vertices are never removed from $\mathcal{S}$—when *find_ap_set* calls *find_ap*, $\mathcal{S}$ is the same as when the last call terminated. Thus it is possible that a scanned edge xy has y in a previous search tree. Examining *find_ap* shows that in this case y is ignored. It is easy to see that properties (i)–(iii) hold for *find_ap_set* as well as *find_ap*.

Now we can show that *find_ap_set* performs as desired.

LEMMA 8.1. *When find_ap_set halts, $\mathcal{P}$ is a maximal set of vertex-disjoint augmenting paths.*

PROOF. By property (ii), $\mathcal{P}$ consists of vertex-disjoint augmenting paths. We need only show that when *find_ap_set* halts, $\mathcal{P}$ is maximal, that is, any augmenting path contains a vertex of $\mathcal{P}$.

When *find_ap_set* halts, consider an alternating path with vertices x_i, $i = 0, \ldots, k$ that starts at a free vertex x_0 and is vertex-disjoint from $\mathcal{P}$. We show by induction that every x_{2j} is outer and $b(x_{2j}) = x_h$ for some $h \leq 2j$. (The argument will also show that the path is not augmenting.)

The base case $j = 0$ holds because *find_ap*(x_0) was called. For the inductive step assume that x_{2j} is outer. Since $x_{2j} \notin \mathcal{P}$ it has been completely scanned, by property (iii). Thus since $x_{2j+1} \notin \mathcal{P}$, x_{2j+1} is not free, that is, x_{2j+1} is matched and either inner or outer. If x_{2j+1} is inner then x_{2j+2} is outer; obviously $b(x_{2j+2}) = x_{2j+2}$ so the inductive assertion holds. (Note that the vertices x_{2j} and x_{2j+1} need not be in the same search tree of *find_ap*.) If x_{2j+1} is outer then it has been completely scanned. Thus $b(x_{2j+1}) = b(x_{2j})$,

from property (i). Since $b(x_{2j}) \neq x_{2j+1}$ by the inductive hypothesis, $b(x_{2j+1}) \neq x_{2j+1}$. Hence x_{2j+2} is outer and $b(x_{2j+2}) = b(x_{2j})$. This completes the induction.

The inductive argument has also shown that no vertex x_{2j+1} is free. This shows there is no augmenting path disjoint from $\mathcal{P}$. □

The time for *find_ap_set* is $O(m)$. To see this, first note that the values of b can be updated and accessed in total time $O(m)$ using the incremental tree set-merging algorithm of [15]. Next note that in a blossom step the vertices u_i are found using the following observation: The vertices u_i' are the predecessors of $b(x)$ in the sequence $(bl)^j b(y)$, $j = 0, \ldots$ (this follows since if v is a blossom base then in the simplified algorithm its grandparent is $b(l(v))$). This implies that in all blossom steps the total time to find all vertices u_i is $O(n)$. It is obvious that the rest of the time for *find_ap_set* is $O(m)$.

After *find_ap_set* the Augment Step augments along each path of $\mathcal{P}$. This takes total time $O(n)$. To see this, note that it is easy to give a recursive routine that finds the edges in a path $P(x)$ in time proportional to their number; after finding an augmenting path it can be augmented. Alternatively, Gabow [8] gives a one-pass procedure.

9. *Analysis Completed: Size of Numbers*

This section completes the efficiency analysis. We have implicitly assumed that all arithmetic operations use $O(1)$ time. To justify this assumption, we show that all numerical values calculated by the algorithm have magnitude $O(n^2 N \log(nN))$. Since the input values require a word size of at least $\max\{\log N, \log n\}$ bits this implies that at worst quadruple-word integers are needed. Thus an aritmetic operation uses $O(1)$ time.

LEMMA 9.1. *At any time in the scaling routine a y or z value has magnitude $O(n^2 N \log(nN))$.*

PROOF. The result is proved in three steps. First we prove it for y values. Define N_s as the largest magnitude of a cost in scale s; it is easy to see that $N_s \leq 2^{s+1} - 2$. Let Y_s denote the largest magnitude of a y value in scale $s \geq 1$, and set $Y_0 = 0$. It suffices to verify the recurrence

$$Y_s \leq 2Y_{s-1} + 1 + 10n \log n + 2nN_s.$$

This implies $Y_s \leq (2^s - 1)(1 + 10n \log n) + ns2^{s+2}$. Hence in the last scale $Y_s = O(n^2 N \log(nN))$. This implies the desired bound for y.

To obtain the recurrence begin by observing that the *match* routine never increases a y value: y values change only in dual adjustments or translations and, if a dual adjustment increases $y(v)$ by δ, the accompanying translation decreases $y(v)$ by δ. Hence it suffices to examine the y values at the end of the scale.

Let ω be the vertex that is free at the end of the scale. We show that at the end of the scale

$$y(\omega) \geq y_0(\omega) - 10n \log n.$$

In *path*(R), suppose some *shell_search*(C, D) adjusts duals by δ. Then $y(\omega)$ does not change if $\omega \in V(C) - V(D)$ and it decreases by 2δ if $\omega \in V(D)$.

Thus the total decrease in $y(\omega)$ is at most the total translation of all even shells, which is at most $5\hat{n}(R)\log \hat{n}(R)$ by (4). Summing over all major path roots R containing ω gives a geometric progression with ratio $1/2$. Thus the total decrease is at most $10n \log n$ as desired.

Now consider any vertex x, with matching M_x at the end of the scale. As in Lemma 2.1, $c(M_x) \leq 2\lfloor n/2 \rfloor + y(V(G)) - y(x) - \lfloor \hat{n}/2 \rfloor z(V(T))$, and $c(M_x) \geq y(V(G)) - y(x) - \lfloor \hat{n}/2 \rfloor z(V(T))$. Thus any vertex x has $y(x) \geq y(\omega) - 2\lfloor n/2 \rfloor + c(M_\omega) - c(M_x)$. Since $c(M_x) \in [-nN_s/2 \,..\, nN_s/2]$, we deduce

$$y(x) \geq y(\omega) - 2nN_s.$$

Combining this with the above inequality for $y(\omega)$ shows that any vertex x has $y(x) \geq y_0(\omega) - 10n \log n - 2nN_s$. The Double Step shows that in scale s, $y_0(\omega)$ has magnitude at most $2Y_{s-1} + 1$. Together these imply the desired recurrence for Y_s.

It remains to analyze the magnitude of z values. Consider first the value $z(G^*)$. This value is nonpositive—it can decrease in Double Steps and translations of *shell_searches*, but it never increases. In some scale s, let e be a matched edge involved in the last blossom step (this step forms the blossom G^* and ends the scale). Since G^* is the only blossom containing e, (1a) implies that $z(G^*) \geq y(e) - N_s$. Thus $z(G^*)$ satisfies the lemma.

Finally consider any z for nonroot blossoms. These values are nonnegative. Consider a nonroot blossom B. Without loss of generality B is a current blossom, so it contains a matched edge e. (1b) for e implies that in scale s, $y(e) + N_s + 2 - z(G^*) \geq z(\{C \mid e \subseteq C\}) - z(G^*) \geq z(B)$. Thus $z(B)$ satisfies the lemma. □

All other quantities computed in the algorithm are easily related to y and z. This includes the quantities of Section 7, z_1, t and y'. (Note that a t value is bounded by a value $z([G^*, B]) - z(G^*)$; the last paragraph of the proof shows that this satisfies the bound of the lemma.)

This completes the analysis of the *scaling routine*.

THEOREM 9.1. *The minimum critical matching problem can be solved in* $O(\sqrt{n\alpha(m, n)\log n}\, m \log(nN))$ *time and* $O(m)$ *space.*

It is interesting that the proofs of the above lemma and Lemma 2.1 use the dual objective function $y(V(G)) - \lfloor \hat{n}/2 \rfloor z(V(T))$. (This is the objective function of the linear programming dual of the matching problem [7].) It is tempting to analyze the matching algorithm using this dual objective function (as done in [10]). Here are some easily-proved facts: The dual objective does not decrease in *path*. In the entire execution of *path* the dual objective increases by $O(n)$ (from the Double Step). A dual adjustment of δ in the search of a shell containing f free vertices increases the dual objective by at least $(f-2)\delta$. These facts give a good bound on the time spent in *shell_searches* of shells with at least three free vertices. Unfortunately it is possible for an even shell to contain only two free vertices. Such shells do not seem amenable to an easy analysis. Hence the attractiveness of this approach remains unclear.

10. *Other Matching Problems*

This section gives applications of the minimum critical matching algorithm.

THEOREM 10.1. *A minimum perfect matching can be found in* $O(\sqrt{n\alpha(m, n)\log n}\, m \log(nN))$ *time and* $O(m)$ *space. The same bounds*

apply to minimum-cost matching and minimum-cost maximum-cardinality matching.

PROOF. The application to minimum perfect matching has been noted in Section 1.1. For minimum-cost matching, observe that a minimum-cost matching on G corresponds to a minimum perfect matching on the graph formed by taking two copies of G and joining each pair of copies of the same vertex by a zero-cost edge. Minimum-cost maximum-cardinality matching uses the same construction except that each added edge costs nN. □

As observed in [10], cost-scaling algorithms can be used as approximation algorithms when input numbers are real, rational or very large integers. We illustrate this with the problem of finding an approximately-minimum perfect matching.

Consider a given cost function c with nonnegative real values. Fix an integer a. We define a new cost function c' with values in $[0\,..\,n^{2+a}]$ such that if M (M') is a minimum perfect matching for c (c'), then $c(M') \leq (1 + n^{-a})c(M)$. To define c' let B be the bottleneck cost of a minimum bottleneck matching; that is, let B be the minimum value such that there is a perfect matching A on the edges costing at most B. Assume $B > 0$; otherwise the problem is trivial. Delete all edges costing more than $c(A)$; clearly these edges are not in a minimum-cost matching. Define cost function $c' = \lfloor n^{1+a}c/B \rfloor$. Note that if e is an edge that is not deleted then $c(e) \leq nB/2$. Hence $c'(e) \leq n^{2+a}$ as desired. Furthermore $c'(M') \leq c'(M)$ implies $n^{1+a}c(M')/B < n^{1+a}c(M)/B + n/2$. Since $c(M) \geq B$, $c(M') < c(M) + B/n^a \leq (1 + n^{-a})c(M)$ as desired.

THEOREM 10.2. *Given arbitrary nonnegative edge costs and a positive integer a, a perfect matching costing at most $(1 + n^{-a})$ times minimum can be found in $O(a\sqrt{n\alpha(m, n)\log n}\, m \log n)$ time and $O(m)$ space.*

PROOF. First a bottleneck matching is found in time $O(\sqrt{n \log n}\, m)$ [14]. Then c' is computed in time $O(m)$. Finally the scaling algorithm is executed with costs c'. This algorithm runs in the time of the theorem. □

This leads to an efficient implementation of Christofides' approximation algorithm for a travelling salesman tour [4]. Recall that this approximation algorithm works as follows. Given are n cities and the distance between each pair of cities. We assume that the distances satisfy the triangle inequality. The algorithm constructs a tour by finding a minimum spanning tree T, finding a minimum perfect matching M on the odd-degree vertices of T, and reducing the Eulerian graph $T \cup M$ to a tour.

Recall the accuracy analysis of this algorithm: Let H denote a minimum-length tour of the given cities. Let $c(e)$ denote the length of an edge e joining two cities. The approximation algorithm gives a tour of length at most $c(T) + c(M)$. It is easy to see that $c(T) \leq (1 - 1/n)c(H)$ and $2c(M) \leq c(H)$. This implies that $c(T) + c(M) \leq (3/2)c(H)$. Hence the algorithm gives a tour at most $3/2$ times optimum.

The running time of this algorithm is $O(n^3)$, the time to find the matching. We improve this by making one change: Instead of M use a perfect matching that is at most $(1 + 1/n)$ times minimum. It is easy to see that the resulting tour is at most $3/2 - 1/(2n) \leq 3/2$ times optimum. We find the approximately-minimum matching using the algorithm of Theorem 10.2.

THEOREM 10.3. *Christofides' approximation algorithm for a travelling salesman tour on n cities, where distances obey the triangle inequality, can be implemented to take* $O(n^{2.5}(\log n)^{1.5})$ *time and* $O(n^2)$ *space.* □

A number of applications of matching require not just an optimum matching but the output of Edmonds' algorithm, an optimum structured matching (recall the definition from Section 1.1). One example is updating a weighted matching: Suppose we have an optimum structured matching and the graph changes at one vertex v (that is, edges incident to v are added or deleted, and costs of edges incident to v change). A new optimum structured matching can be found in the time for one search of Edmonds' algorithm [2, 5, 10, 29]. Another example is the single-source shortest path problem on undirected graphs with no negative cycles [21, pp. 220–222]. We now give an algorithm to find an optimum structured matching.

The algorithm starts by executing the *scaling routine* with one change: The new cost function $\bar{c}$ is $(2n+2)c$ (in Section 2, $\bar{c} = (n+1)c$). Change the number of scales correspondingly to $k = \lfloor \log(n+1)N \rfloor + 2$. Suppose the *scaling routine* halts with matching M_x, blossom tree T and dual functions y_0, z_0. Our structured matching has the same matching and blossom tree. The dual function y is defined by

$$Y = y_0(V(G)) - \lfloor \hat{n}/2 \rfloor z_0(V(T));$$
$$y = \left\lfloor \frac{y_0 - Y}{2n+2} \right\rfloor.$$

To define z, for each blossom B choose (arbitrarily) a blossom edge e_B of B. For a blossom B with parent A,

$$z(B) = (y-c)(e_B) - (y-c)(e_A);$$
$$z(G^*) = (y-c)(e_{G^*}).$$

To prove that the algorithm is correct, define dual functions $\bar{y} = (2n+2)y$, $\bar{z} = (2n+2)z$. It suffices to show that changing the duals to $\bar{y}$, $\bar{z}$ gives an optimum structured matching for the cost function used by the *scaling routine*, $\bar{c}$. This amounts to showing the following: (i) the duals are tight on every blossom edge; (ii) the duals are dominated on every edge; (iii) $\bar{z}(B) \geq 0$ unless $B = G^*$. The following proof depends on the fact that all values of $\bar{y}$, $\bar{z}$ and $\bar{c}$ are multiples of $2n+2$.

We start by showing that for any vertex v,

$$y_0(v) - \bar{y}(v) - Y \in [0\,..\,n). \tag{6}$$

First observe that the quantity $y_0(v) - Y + \bar{c}(M_v) \in [0\,..\,n)$, by (1a)—(1b) applied to the edges of M_v. Furthermore $\bar{c}(M_v)$ is a multiple of $2n+2$. Since by definition $\bar{y}(v) = (2n+2)\lfloor (y_0(v) - Y)/(2n+2) \rfloor$, (6) follows.

To prove (i)–(ii) we first show that any edge e satisfies the relation

$$(\bar{y} - y_0)e - (\bar{z} - z_0)([G^*, B]) \in [-2n+2\,..\,2n].$$

To derive this, first observe the tightness equation for any edge e_B, $\bar{y}\bar{z}(e_B) = \bar{c}(e_B)$ (this follows from the definition of z). From (1a)–(1b), $y_0 z_0(e_B) - \bar{c}(e_B) \in [-2\,..\,0]$. Subtract the previous tightness equation and use (6) on the

ends of e_B to get $(\bar{z} - z_0)([G^*, B]) + 2Y \in [-2n \,..\, 0]$. The desired relation for e follows by using this last relation plus (6) on the ends of e.

To prove (i), consider any blossom edge e of B. Since $y_0 z_0(e) - \bar{c}(e) \in [-2 \,..\, 0]$, the above relation for e implies that $\bar{y}\bar{z}(e) - \bar{c}(e) \in [-2n \,..\, 2n]$. Since the left-hand expression is divisible by $2n + 2$, it equals zero, that is, the duals are tight on e. For (ii), similarly consider an edge e such that B is the smallest blossom containing it. Since $y_0 z_0(e) \leq \bar{c}(e)$ it is easy to see that $\bar{y}\bar{z}(e) - \bar{c}(e) \leq 2n$. Since the left-hand expression is divisible by $2n + 2$, $\bar{y}\bar{z}(e) \leq \bar{c}(e)$, that is, the duals are dominated on e.

Lastly consider (iii). From (1a)–(1b), $(y_0 - \bar{c})e_B - (y_0 - \bar{c})e_A - z_0(B) \in [-2 \,..\, 2]$. Combining this with the definition of $\bar{z}$ shows that $(\bar{z} - z_0)(B) + (y_0 - \bar{y})e_B - (y_0 - \bar{y})e_A \in [-2 \,..\, 2]$. Applying (6) to the ends of e_B and e_A implies that $(\bar{z} - z_0)(B) \in [-2n \,..\, 2n]$. This implies that $\bar{z}(B) \geq -2n$, since $z_0(B) \geq 0$. This gives the desired relation $\bar{z}(B) \geq 0$, since $\bar{z}(B)$ is divisible by $2n + 2$.

THEOREM 10.4. *An optimum structured matching can be found in the time and space bounds of Theorem* 9.1.

The minimum critical matching algorithm can be modified to find a maximum-cardinality matching on an arbitrary graph G. The cardinality matching algorithm works as follows. Define all edge costs to be zero. Execute the *scaling routine* by simply omitting the Double Step and doing the Match Step once, that is, call the *match* routine once to find the desired matching. Define ρ, the number of phase 1 iterations of *path*, to be $\lceil \sqrt{n} \rceil$. The remaining details of phase 1 are unchanged. (The algorithm works with one shell, G^*, which is even or odd depending on the number of vertices of G.) After phase 1 the algorithm is simpler than before. Instead of phases 2–3 it abandons all edge costs and dual variables and does the following: Repeatedly call *find_ap_set* to find a maximal set of disjoint augmenting paths $\mathscr{P}$, and augment along these paths. *find_ap_set* operates on the graph G, unmodified. The algorithm halts when *find_ap_set* does not find an augmenting path.

The analysis of this algorithm is a special case of critical matching. We sketch it for completeness. First recall that the old blossom tree T has root G^* with children $V(G)$. The following version of Lemma 6.1 holds: At any time in the execution of *path*(G^*) let M be the current matching. Let M_0 be a maximum-cardinality matching whose free vertices are all free in M. Let F_ω be the set of vertices that are free in M but not M_0.

LEMMA 10.1. *At any time in path*(G^*) *during phase* 1 *of the cardinality matching algorithm*, $\hat{\delta}(F_\omega, G^*) \leq n$.

PROOF. The proof is a special case of Lemma 6.1. As in Lemma 6.1, define $\mu(B) = |M_0 \cap G(B)| - |M \cap G(B)|$. For the current duals y, z, adding (1a) for M_0 and subtracting (1b) for M, and recalling that all edge costs are zero, gives

$$y(F_\omega) - \mu z(\{G^*\} \cup V(T)) \leq 2|M|.$$

Note that y is zero for any free vertex and μ is nonpositive for a current blossom (as in Lemma 6.1). Hence $-\mu z(G^*) \leq 2|M|$. Now the relations $z(G^*) = -2\hat{\delta}(G^*)$, $\mu(G^*) = |F_\omega|/2$, and $|M| \leq n/2$ imply the lemma. □

At the end of phase 1 any free vertex v has $\hat{\delta}(v, G^*) = \rho$. Thus $|F_\omega| \rho \leq n$. This implies that phase 1 ends with $O(\sqrt{n})$ more free vertices than a maximum-cardinality matching. Thus *find_ap_set* is executed $O(\sqrt{n})$ times.

As in the weighted algorithm, the time for a phase 1 iteration and the time for one execution of *find_ap_set* are both $O(m)$. Thus the total time is $O(\sqrt{n}\, m)$.

THEOREM 10.5. *A maximum cardinality matching can be found in* $O(\sqrt{n}\, m)$ *time and* $O(m)$ *space.*

This bound is the same as the algorithm of Micali and Vazirani [23]. Note that our algorithm is not the same as theirs. For instance it operates with inner blossoms effectively "shrunk" during phase 1. Also our depth-first search seems to involve less overhead than the "double depth-first search" of [23].

In practice a different organization after phase 1 is probably faster: The algorithm initializes the search graph $\mathcal{S}$ to empty. Then it calls *find_ap*(f) for each free vertex f. If *find_ap*(f) finds an augmenting path then the matching is augmented; also the vertices that it added to $\mathcal{S}$ are removed from $\mathcal{S}$. If *find_ap*(f) does not find an augmenting path then nothing is done—the vertices it added to $\mathcal{S}$ remain in $\mathcal{S}$. These vertices will be effectively ignored in subsequent searches. This is correct because these vertices cannot be in augmenting paths (arguing as in Lemma 8.1; alternatively see [6] and [8].

11. *Concluding Remarks*

The matching algorithm generalizes to degree-constrained subgraph problems. Consider a graph with two functions $l, u: V \to \mathbf{Z}$. A *degree-constrained subgraph* (*DCS*) is a subgraph in which each vertex v has degree in the range $[l(v) \,..\, u(v)]$. In a *perfect DCS* each degree is exactly $u(v)$. The size of a perfect DCS is denoted $U = u(V)$. The *weighted degree-constrained subgraph problem* is to find a minimum-cost maximum-cardinality DCS or a minimum-cost DCS. A degree-constrained subgraph problem on a graph of n vertices and m edges can be reduced in linear time to a matching problem on a graph of $O(m)$ vertices and edges [11]. Thus our algorithm immediately implies a bound of $O(\sqrt{m\alpha(m, m)}\log m\, m \log(mN))$ to solve the weighted DCS problem. The same bound applies to the problem of finding a minimum cost flow on a 0–1 bidirected network [21]. A more careful implementation of our ideas gives a bound of $O(\sqrt{U\alpha(m, n)}\log U\, m \log(nN))$ for the weighted DCS problem; this bound holds for multigraphs as well. Details will be given in a forthcoming paper.

Pravin Vaidya has recently investigated the matching problem for points in the plane. If distance is measured by the L_1, L_2 or L_∞ norm, a minimum perfect matching on a set of $2n$ points can be found in time $n^{2.5}\log^{O(1)} n$ and space $O(n \log n)$ [28]. Furthermore it appears that applying our algorithm reduces the time by a factor of about $\sqrt{n}$ [28].

REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. BALL, M. O., AND DERIGS, U. An analysis of alternative strategies for implementing matching algorithms. *Networks 13*, 4 (1983), 517–549.
3. BERTSEKAS, D. P. Distributed asynchronous relaxation methods for linear network flow problems, LIDS Report P-1606, M.I.T., Cambridge, Mass., 1986; preliminary version. In *Proceeding of the 25th Conference on Decision and Control*. IEEE, New York, December 1986.

4. CHRISTOFIDES, N. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon Univ., Pittsburgh, Pa., 1976.
5. CUNNINGHAM, W. H., AND MARSH, A. B. III. A primal algorithm for optimum matching. *Math. Prog. Stud. 8* (1978), 50–72.
6. EDMONDS, J. Paths, trees and flowers. *Canad. J. Math. 17* (1965), 449–467
7. EDMONDS, J. Maximum matching and a polyhedron with 0, 1-vertices. *J. Res. Nat. Bur. Standards 69B* (1965), 125–130.

7a. EDMONDS, J., AND KARP, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM 19*, 0 (1972), 248–264.

8. GABOW, H. N. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *J. ACM 23*, 2 (1976), 221–234.
9. GABOW, H. N. Scaling algorithms for network problems. *J. Comp. and System Sci., 31*, 2 (1985), 148–168.
10. GABOW, H. N. A scaling algorithm for weighted matching on general graphs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*. IEEE, New York, 1985, pp. 90–100.
11. GABOW, H. N. Duality and parallel algorithms for graph matching, manuscript.
12. GABOW, H. N. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 1990, pp. 434–443.
13. GABOW, H. N., GALIL, Z., AND SPENCER, T. H. Efficient implementation of graph algorithms using contraction. *J. ACM 36*, 3 (1989), 540–572.
14. GABOW, H. N., AND TARJAN, R. E. A linear-time algorithm for a special case of disjoint set union. *J. Comp. and System Sci., 30*, 2 (1985), 209–221.
15. GABOW, H. N., AND TARJAN, R. E. Algorithms for two bottleneck optimization problems. *J. Algorithms, 9*, 3 (1988), 411–417.
16. GABOW, H. N., AND TARJAN, R. E. Faster scaling algorithms for network problems. *SIAM J. Comput., 18*, 5 (1989), 1013–1036.
17. GALIL, Z., MICALI, S., AND GABOW, H. N. An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM J. Comput., 15*, 1 (1986), 120–130.
18. GOLDBERG, A. V. Efficient graph algorithms for sequential and parallel computers. Ph. D. Dissertation, Dept. of Electrical Eng. and Comp. Sci., MIT, Technical Rep. MIT/LCS/TR-374, Cambridge, Mass., 1987.
19. GOLDBERG, A. V., AND TARJAN, R E. Finding minimum-cost circulations by successive approximation. *Math. of Oper. Res., 15*, 3 (1990), 430–466.
20. HOPCROFT, J., AND KARP, R. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, *SIAM J. Comput., 2*, 4 (1973), 225–231.
21. LAWLER, E. L. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.
22. LOVÁSZ, L., AND PLUMMER, M. D. *Matching Theory* North-Holland Mathematical Studies 121, Annals of Discrete Mathematics 29, North-Holland, New York, 1986.
23. MICALI, S., AND VAZIRANI, V V. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual Symposium on the Foundations of Computer Science*. IEEE, New York, 1980, pp. 17–27.
24. PAPADIMITRIOU, C. H., AND STEIGLITZ, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc , Englewood Cliffs, N.J., 1982.
25. TARDOS, É. A strongly polynomial minimum cost circulation algorithm. *Combinatorica 5*, 3 (1985), 247–255.
26. TARJAN, R. E. Applications of path compression on balanced trees. *J. ACM 26*, 4 (1979), 690–715.
27. TARJAN, R. E. *Data Structures and Network Algorithms*. SIAM Monograph, Philadelphia, Pa., 1983.
28. VAIDYA, P. M. Geometry helps in matching In *Proceeding of the 20th Annual ACM Symposium on Theory of Computing*. ACM, New York, 1988, pp 422–425.
29. WEBER, G. M. Sensitivity analysis of optimal matchings, *Networks 11* (1981), 41–56.

RECEIVED APRIL 1989; REVISED MAY 1990; ACCEPTED MAY 1990