FASTER SCALING ALGORITHMS FOR GENERAL GRAPH MATCHING PROBLEMS

Harold N. Gabow
Robert E. Tarjan

CS-TR-222-89

April 1989

Faster Scaling Algorithms for General Graph Matching Problems

Harold N. Gabow[1]
Department of Computer Science
University of Colorado
Boulder, CO
80309

Robert E. Tarjan[2]
Computer Science Department
Princeton University
Princeton, NJ 08544
and
AT&T Bell Laboratories
Murray Hill, NJ 07974

April 11, 1989

Abstract. This paper presents an algorithm for minimum cost matching on a general graph with integral edge costs, that runs in time close to the best known bound for 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 the same time as above, $O(\sqrt{n}m)$. 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.

[1] Research supported in part by NSF Grant Nos. DCR-851191, CCR-8815636, and AT&T Bell Laboratories.

[2] Research supported in part by NSF Grant No. DCR-8605962 and ONR Contract No. N00014-87-K-0467.

1. Introduction.

The problem of finding a minimum cost matching on a general graph is a classical problem in network optimization, with many practical applications and very efficient algorithms. We present an algorithm for this problem that is almost as fast as the best known algorithm for the problem without costs, 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 associated numeric values (costs, lengths, etc.) and the values are integral, N denotes the largest magnitude of a value.

The best known algorithm for maximum cardinality matching is due to Micali and Vazirani [MV] and runs in time $O(\sqrt{n}m)$ (see also [GT85]). Edmonds gave the first polynomial algorithm for weighted matching [E65b]. The best known implementation of this algorithm 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 [GGS]. This bound can be substantially improved under the assumption of integral costs that are not huge: The scaling algorithm of [G85b] 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 above algorithm of Micali and Vazirani. We present two other applications: We show how to speed up Christofides' travelling salesman approximation algorithm [C] to $O(n^{2.5}(\log n)^{1.5})$; this bound is independent of the size of the imput numbers. We also show how to find the linear programming duals for matching, that are the basis of Edmonds' algorithm. This gives efficient solutions of 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 [Go, GoT87a-b], and applied in [GT87] to the assignment problem. The first approaches to scaling computed an optimum solution at each of $\log N$ scales (e.g., [EK], [Ga85a-b]). 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 [Ber86, Tard] 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 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 [G85b], including the notion of "shells", are used to overcome this difficulty. Nonetheless blossoms slow our algorithm down: The algorithm of [GT87] 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 blossom manipulation.

The paper is organized as follows. Section 1.1 reviews Edmonds' weighted matching algorithm [E65b]; 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 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 gives surveys further applications of the algorithm.

This 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 [L, T83].

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, $[i..j] = \{k | k \text{ is an integer}, i \le k \le j\}$. The function $\log n$ denotes logarithm to the base two.

We use a hat, e.g., $\widehat{f}$, to emphasize that an object is a function. 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, i.e., $(f+g)(x) = f(x)+g(x)$, $fg(x) = f(x) \times g(x)$. 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 $f(S) = \sum\{f(s) | s \in S\}$. Similarly if f has two arguments then $f(S,T) = \sum\{f(s,t) | s \in S, t \in T\}$, for $S \times T$ a subset of the domain of f.

For a graph G, $V(G)$ and $E(G)$ denote the vertex set and the edge set, respectively. The given graph G has n vertices and m edges. 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 both vertices of e are in S. We usually denote the edge joining vertices v and w by vw. Thus if $e = vw$ and $y : E(G) \to \mathbf{R}$ 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.

Suppose $c : E \to \mathbf{R}$ is a function that assigns a numeric *cost* to each edge; in this paper costs are integers in $[-N..N]$ 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 1.1 shows an x-matching; in all figures of this paper matched edges are drawn wavy and free vertices are drawn square. A *minimum* (*maximum*) *v-matching* is a v-matching with minimum (maximum) possible cost. G is *critical* if every v has a v-matching. The *minimum critical matching problem* is: given a critical graph, find a minimum v-matching for each vertex v [G87]. It follows from [E65a] that all the desired matchings can be represented the blossom tree defined below; we shall accept the 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 that satisfies the following properties. (Figure 1.2 shows a blossom forest, with just one tree, for the graph of Figure 1.1). 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 $V(B) = \cup_{i=1}^{k} V(B_i)$ and $E(B) = \cup_{i=1}^{k}(E(B_i) + e_i)$, 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 the edges are both unmatched, and for all other children one edge is matched and the other unmatched; there are precisely two possible orderings of the children). In this 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, i.e., 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

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

We emphasize that a blossom is not defined as an induced subgraph, e.g., $\widehat{m}(B)$ need not equal $|E(B)|$. A simple induction shows that $\widehat{n}(B)$ is odd. The *base vertex* of B is the unique vertex of B not on a matched edge of $E(G(B))$. The base of a vertex v is v; a simple induction shows 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 [E65a].) 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, $1 \leq i \leq k$; as above, e_i joins B_i to B_{i+1}. Let $w \in B_j$. Match alternate edges of the list e_i, $i = j, j+1, \ldots, k, 1, 2, \ldots, j-1$, keeping the first and last edges e_j and e_{j-1} unmatched. For $i \neq j$ let w_i denote the vertex of B_i on a matched edge of this list; 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 [E65b]. Two functions y, z form (a pair of) *dual functions* if $y : V(G) \rightarrow \mathbf{R}$, $z : 2^{V(G)} \rightarrow \mathbf{R}$ and $z(B) \geq 0$ unless $B = V(G)$. Such a pair determines a *dual edge function* $\widehat{yz} : E \rightarrow \mathbf{R}$ which for an edge e is defined as

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

(Recall that by convention if $e = vw$ then $y(e) = y(v) + y(w)$.) The duals are *dominated* on edge e

if

$$\widehat{yz}(e) \leq c(e);$$

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. It is easy to see that 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 [E65b] halts; see below).

More precisely a *search* builds a *search graph* $\mathcal{S}$, defined as follows and illustrated in Figure 1.3. $V(\mathcal{S})$ 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 $\mathcal{F}$. The roots of $\mathcal{F}$ are precisely the root blossoms of G that contain a free vertex. A root blossom of $\mathcal{S}$ is *outer* if its distance (in $\mathcal{F}$) from a root of $\mathcal{F}$ is even, or *inner* if its distance is odd. Any descendant of an outer (inner) root blossom is also called outer (inner). (Hence every free vertex of G is outer.) Any outer vertex v is joined to a free vertex by an even length alternating path $P(v) \subseteq E(\mathcal{S})$.

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

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: first an unmatched edge e is added, along with the above blossom B; then the matched edge incident to B is added. Figure 1.4 shows a grow step for the unmatched edge ab followed by a grow step for the matched edge cd.

A *blossom step* enlarges $\mathcal{S}$ by adding a tight edge that joins distinct outer root blossoms of $\mathcal{S}$. This step either constructs a new blossom in $\mathcal{S}$, or it discovers that $\mathcal{S}$ contains an augmenting path. In Figure 1.3 an edge ae would give a blossom step that constructs a new blossom, possibly the one in Figure 1.1.

An *expand step* deletes an unweighted root blossom B from the blossom forest, thus making its children into roots; B is also deleted from $\mathcal{S}$ and replaced by some (but not necessarily all) of these children, so that $\mathcal{F}$ remains a forest. Figure 1.5 shows an expand step; blossoms B_i become root blossoms, and B_4 and B_5 leave $\mathcal{S}$.

These three steps are repeated as many times as possible, giving a maximal search graph $\mathcal{S}$. If the maximal $\mathcal{S}$ 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:

$$y(v) \leftarrow y(v) + \delta, \text{ for each outer vertex } v;$$

$$y(v) \leftarrow y(v) - \delta, \text{ for each inner vertex } v;$$

$$z(B) \leftarrow z(B) + 2\delta, \text{ for each nonleaf root outer blossom } B;$$

$$z(B) \leftarrow z(B) - 2\delta, \text{ for each nonleaf root inner blossom } B.$$

These assignments do not change the value of $\widehat{yz}(e)$ for $e \in E(\mathcal{S})$, so these edges remain tight. The assignments increase $\widehat{yz}(e)$ only if e joins an outer vertex to a vertex not in $\mathcal{S}$, or if e joins two distinct root outer blossoms. Thus the adjustment maintains condition (*ii*) above and also allows a new grow, blossom or expand step to be executed, if δ is chosen as $\delta = \min\{\delta_g, \delta_b, \delta_e\}$ where

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

$$\delta_b = \min\{(c-y)(e)/2 \mid e \text{ an edge joining two distinct root outer blossoms}\};$$

$$\delta_e = \min\{z(B)/2 \mid B \text{ a nonleaf root inner blossom}\}.$$

Note that $\delta > 0$. 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 final details of the dual adjustment for use below. First, the dual $y(v)$ of any free vertex v increases by δ. Second, note that the dual adjustment step does divisions by two to calculate δ_b. If all given costs are even integers then all quantities computed by the algorithm are integers [PS, p. 267, ex. 3]. This fact motivates various details of the scaling algorithm, which keeps edge costs even for this reason. (These details are all noted below.)

2. The matching algorithm: the *scaling* routine.

This section gives the overall structure of the new 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, that has almost minimum cost, 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:

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

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

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 [GT87]. (Actually the bipartite matching algorithm of [GT87] 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, $k \geq n$, 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 $\lfloor \widehat{n}(B)/2 \rfloor$ edges of $G(B)$, and no matching contains more. Combining these facts with ($1a$) – ($1b$) gives

$$c(M_x) \leq 2\lfloor n/2 \rfloor + \widehat{yz}(M_x) \leq 2\lfloor n/2 \rfloor + y(V(G) - x) - \lfloor \widehat{n}/2 \rfloor z(V(T)) \leq c(X) + n - 1.$$

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

(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. The algorithm always works with even edge costs to preserve integrality. The *scaling routine* starts by computing a new cost function $\overline{c} = (n+1)c$ (thus each $\overline{c}(e)$ is even). It maintains a cost function c equal to $\overline{c}$ in the current scale. Define $k = \lfloor \log{(n+1)N} \rfloor + 1$, the greatest number of bits in the binary expansion of a $\overline{c}$ cost. For any $s \in [1..k]$ define a function $b_s : [-(n+1)N..(n+1)N] \rightarrow \{-1, 0, 1\}$ by taking $b_s(i)$ as the s^{th} signed bit in the expansion of i as a k-bit binary number. For example any edge e has $b_k(\overline{c}(e)) = 0$. The *scaling routine* initializes c, y and z to the zero function, the matching M_x to $\emptyset$, 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, i.e., 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 (at the end of Section 4) that the algorithm still works correctly, because $z = 0$.

3. The *match* routine.

This section describes the overall structure of the routine that finds a 2-optimal 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 also 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 [GT87]. In the bipartite algorithm, each scale is similar to the first in that the dual function can be taken to be 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, which cannot be eliminated (see [G85b]). 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. Note that both current and old blossoms contribute to the z term in the definition of $\widehat{yz}(e)$. When all old blossoms are dissolved, the matching is 2-optimum and the routine can halt. The reason is that 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 $\widehat{n}(C)$; 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 $\widehat{n}(D) < \widehat{n}(B)/2$. 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 [T79].) A major path starting at vertex R is denoted by $P(R)$ and has ***major path root*** R. Define the ***rank*** of any node B as $\lfloor \log \widehat{n}(B) \rfloor$. 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 1.2 the path from the root to leaf z is the major path $P(G^*)$; the root has rank 4.

procedure *match.*
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 $R = G^*$, it halts with a 2-optimum matching, as noted above. Note that for any major path root R, on entry to $path(R)$ all descendants of R have dissolved except those on $P(R)$. Figure 3.1 illustrates the *match* routine: Suppose the graph of Figure 1.1 has old blossom tree given by Figure 1.2. 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 $O(\sqrt{\widehat{n}(R)\alpha(m,n)\log \widehat{n}(R)}\ \widehat{m}(R))$ then the time for *match* is $O(\sqrt{n\alpha(m,n)\log n}\ m)$.

Proof. For any integer $0 \le r \le \log n$, 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 [G85b].

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 $D = \emptyset$. Extend the function $\widehat{n}$ to shells: $\widehat{n}(C,D)$ is the number of vertices in a shell $G(C,D)$. A shell is *even* if $\widehat{n}(C, D)$ is even, or equivalently, $D \neq \emptyset$; otherwise it is *odd*. Figure 3.1 indicates the even shell $G(G^*, A)$.

We use a number of functions of shells, like the above $\widehat{n}$. We define such functions by using the shell boundaries as arguments, as in the above $\widehat{n}(C, D)$. Alternatively if X denotes a shell we use X as the argument, e.g., $\widehat{n}(X)$. On the other hand if X denotes an old blossom it corresponds to an odd shell, and we write $\widehat{n}(X)$ as a shorthand for $\widehat{n}(X, \emptyset)$. Which of the two interpretations of $\widehat{n}(X)$ 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. Clearly these shells change as blossoms dissolve.

The *path*(R) routine works in a manner similar to the bipartite matching algorithm of [GT87], in the sense that it finds practically all the augmenting paths immediately, and then finds the remaining paths at a slower and slower rate. The bipartite algorithm accomplishes this automatically, i.e., 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 $P(R)$ (clearly 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, i.e., 1; 1,2; 1,2,3; or 1,3.

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 $\widehat{yz}(e) = c(e)$;

(ii) e is matched and $\widehat{yz}(e) = c(e) - 2$;

(iii) e is a current blossom edge.

In phase 2 or 3, the condition is $\widehat{yz}(e) \in [c(e) - 2..c(e)]$. Note that this is always the case if any of

(i) – (iii) hold.

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 $P(R)$ dissolve:

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, $i = 1, \cdots, k$ so that $\widehat{n}(S_i)$ 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 (see below). ▌

The *path* routine is implemented as follows. If R is a leaf blossom, *path* exits immediately (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 Section 8 does the depth-first search in time $O(\widehat{m}(R))$.

In the Search Step, *shell_search* is the *search* step 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 1.4, 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 (the duals of) B by δ* means to perform the following assignments:

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

$$z(B) \leftarrow z(B) - 2\delta.$$

(For example a dual adjustment in Edmonds' algorithm translates inner root blossoms.) Observe that translating an undissolved blossom B cannot increase a quantity $\widehat{yz}(e)$, and it maintains inequalities (1) ((1b) holds since no such edge e has exactly one vertex in B).

Consider an undissolved shell $G(C, D)$ containing a free vertex. The routine *shell_search*(C, D) executes a *search* step of Edmonds' algorithm on $G(C, D)$, modified to translate C and D. More precisely when the *search* step 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 $\delta = \delta_d$, i.e., the translation dissolves C or D (or both). The dissolve step 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 $G(C', D')$.

Note that translating C ensures inequality (2) is preserved, since *search* increases a y-value by at most δ. (2) in turn ensures (1a) for edges going out of the blossom C. The translation of D is needed to preserve (1a) on edges vw, $v \in V(D)$, $w \in V(C) - V(D)$: translating C decreases $z(C)$ by 2δ and $y(w)$ may have no net change, so $y(v)$ may need to decrease by 2δ. We conclude that each step of *shell_search* preserves (1) (on all edges of G) and (2).

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 G is a blossom. These rules are also used in *shell_search*, but they are in fact subsumed by new halting criterion. We discuss the new halting rules for phases 3,1 and 2, in that order.

In phase 3 by definition no augmenting path can be found in R. *shell_search* halts when all blossoms on $P(R)$ are dissolved.

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 stops. 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. The *shell_search* routine removes any unweighted root of the current blossom forest and replaces it by its children, until every nonleaf root is weighted.

In phase 2 *shell_search*(S) can halt in three different ways. It halts immediately if S is an odd shell that does not contain all the free vertices of R'. Otherwise *shell_search* halts when it finds an augmenting path of eligible edges. As a special case, *shell_search*(S) halts if a dissolve step enlarges S by adding a new shell S', where S' has already been searched (in the current Search Step) and found to contain an augmenting path.

The third way to halt 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 $R = G^*$.)

This concludes the statement of *path*. We now summarize some more facts that further motivate and justify the phase structure.

Details of *shell_search* for the three phases are given in Section 7. For the analysis of next section we only need the following summary. It indicates that, as already mentioned, the Search Step consumes more time in later phases. The usual implementation of *search* in Edmonds' algorithm (e.g., [GGS]) 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 $O(\widehat{m}(R))$. In phase 2, the priority queue can be implemented as an array. The phase 2 Search Steps use total time $O(\widehat{n}(R)\log n)$ to scan the array; in addition, each Search Step uses $O(\widehat{m}(R)\alpha(m,n))$ time. In phase 3 a standard priority queue is used; the time for phase 3 is (less than) $O(\widehat{m}(R)\log n)$.

Next we indicate why the definition of "eligible" changes for phases 2–3. Recall that in *search*, if a dual adjustment makes an inner blossom B unweighted an *expand* step is done. This step removes B, a root, from the blossom forest, making its children into root blossoms. In $\mathcal{S}$, B is replaced by an alternating path of edges of $E(B)$. This enables the portion of $\mathcal{S}$ descending from B to remain in $\mathcal{S}$. (See Figure 1.5).

Recall that such *expand* steps do not occur in phase 1, since a phase 1 search stops right after its dual adjustment. *Expand* steps do occur in phases 2–3. Now observe that an edge $e \in E(B)$ that gets placed in $\mathcal{S}$ in the *expand* step may not satisfy alternatives (i) — (iii) of eligibility. The 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 $\mathcal{S}$ (see Figure 1.5). At this time e is an unmatched edge in $\mathcal{S}$ with $\widehat{yz}(e) = c(e) - 2$, and e is no longer a blossom edge. Thus e does not satisfy any of alternatives (i) – (iii) of eligibility.

To remedy this, phases 2–3 use the weaker definition of eligibility. This makes the above edge e eligible. The weaker definition of eligibility suffices for these phases. (However the weaker definition would not be adequate for phase 1, see Lemma 5.1.)

This concludes the description of *path*. Showing *path* is correct amounts to checking 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 $\widehat{yz}(e) \geq c(e) - 2$ and hence satisfies ($1b$) if it 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 one inequality. This inequality is derived in the next section, thereby completing the analysis.

We start with 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 either by adjusting the duals or augmenting the matching. Any phase 2 iteration augments the matching. (However it need not adjust duals, because of the different definition of eligiblity.) A phase 1 iteration may not augment the matching, as remarked above. We now show that any phase 1 iteration adjusts the duals. This statement is true in spite of the fact that there may exist an augmenting path of eligible edges when *shell_search* is called (the path can be hidden by blossoms).

Lemma 5.1. In phase 1 any execution of $shell\_search(S)$ adjusts duals by $\delta = 1$.

Proof. 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. We first observe that P corresponds to an augmenting path in the auxiliary graph H of the preceding Augment Step: Call the preceding Augment Step $\mathcal{A}$. Clearly the edges of P were eligible in $\mathcal{A}$, since the duals were not adjusted. Now consider a root blossom B that was contracted to form H. It suffices to show if $V(P) \cap V(B)$ is nonempty then it is a subpath of P containing the base vertex of B. For this it suffices to show that B is a blossom at the end of *shell_search* (since the desired property holds for any blossom at the end of *shell_search*). Before doing a dual adjustment *shell_search* can do *grow* and *blossom* steps, but no *expand* steps. Hence any blossom B that was current in $\mathcal{A}$ is current when *shell_search* halts (although B may be included in a new, larger blossom). This gives the desired conclusion.

The definition of the Augment Step implies that P (considered as a path in H) contains a vertex of some path $Q \in \mathcal{P}$. P does not contain an edge of $E(Q) \cap E(H)$, since these edges become ineligible when Q gets augmented. (Here we use the hypothesis that the algorithm is in phase 1, not 2.) Thus P contains a vertex in a current blossom B on Q. As noted above, this implies that P contains the base vertex of B (after Q is augmented). Thus P contains the matched edge e incident to B. But e became ineligible in the augment of Q. 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 let $n = \widehat{n}(R)$ and $m = \widehat{m}(R)$. Recall ρ is the number of iterations of the loop of *path* in phase 1. At the end of phase 1 let R_1 be the largest undissolved blossom in $P(R)$. Let F_1 denote the set of free vertices of R_1. The number of phase 2 iterations is at most $|F_1|/2$ since every phase 2 iteration augments the matching. Assume for the moment that this *product inequality* holds:

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

Thus if $\rho \geq 2 \log n$ then the number of phase 2 iterations is at most $(5n \log n)/\rho + 1/2$.

Recall the time bounds for the various phases, as already mentioned and presented in detail in Section 7: $O(m)$ for one iteration in phase 1, $O(m\alpha(m,n))$ for one iteration in phase 2 plus $O(n \log n)$ total extra time, and $O(m \log n)$ total time for phase 3. Thus the total time for *path*(R) is $O(\rho m + ((n \log n)/\rho) m\alpha(m,n) + m \log n)$. Taking

$$\rho = \max\{\sqrt{n\alpha(m,n)\log n}, 2\log n\}$$

gives time $O(\sqrt{n\alpha(m,n)\log n}\ m)$ for *path*(R). Then 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". To state the latter 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

$$\begin{aligned} \widehat{\delta}(B) &= \text{ the total of all translations of } B; \\ \widehat{\delta}(v,B) &= \text{ the total of all translations of } B \\ &\qquad \text{made when } v \text{ is in an undissolved shell with outer boundary } B. \end{aligned}$$

Note that there can be more than one inner boundary of shells contributing to $\widehat{\delta}(B)$ and $\widehat{\delta}(v,B)$. Since translating B by one decreases $z(B)$ by two, $\widehat{\delta}(B) \le z_0(B)/2$; equality holds if B dissolves before becoming a current blossom. The quantity $\widehat{\delta}(v,B)$ counts all translations of B "witnessed by" v; $\widehat{\delta}(v,B) > 0$ only when $v \in B$.

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

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

(This inequality is one reason why we perform the analysis in terms of $\widehat{\delta}$ rather than quantities y, z directly involved in the algorithm. Intuitively $\widehat{\delta}(v, \cdot\,)$ 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 $y(v)$ can change even though no progress is being made for v — specifically in executions *shell_search*(A,B) where $v \in B$ (see 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 executes *shell_search*$(S(v))$ in all but at most $\log n$ iterations of *path*. The Search Step does not execute *shell_search*$(S(v))$ 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 $\widehat{n}(S(v))$ doubles. Since this can happen only $\log n$ times the desired conclusion follows.

Every phase 1 execution of *shell_search*$(S(v))$ adjusts duals by $\delta = 1$ (Lemma 5.1). Thus in ρ iterations of phase 1, v witnesses at least $\rho - \log n$ translations, i.e., $\widehat{\delta}(v, P(R)) \ge \rho - \log n$, and $\widehat{\delta}(F-\omega, P(R)) \ge (\rho - \log n)(|F_1| - 1)$. This plus the witness inequality obviously implies the product inequality.

6. The witness inequality.

This section derives the witness inequality, thereby completing the efficiency analysis for *path*. It also derives a related inequality needed for *shell_search*.

We start with terminology. We use an interval notation for paths in trees: 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 $P(R)$. Recall that G^* is the root of 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, $\mathcal{DN}(P)$ denotes the set of all descendants of nonmajor children of nodes of P. If P is an interval omit the enclosing parentheses in these notations, so $\mathcal{N}[C, D]$ and $\mathcal{N}[C, D)$ have the obvious meanings.

The derivation concentrates on three types of shells $G(C, D)$. In each type $G(C, D)$ is a shell of $P(R)$. $G(C, D)$ is *original* if C is the parent of D (an odd shell C is original if C is the last nonleaf in $P(R)$). $G(C, D)$ is *active* if it is even and C and D both dissolve after each blossom in (C, D). For example an even original shell is active. Also if the Search Step executes *shell_search*(C, D) where $D \neq \emptyset$, then $G(C, D)$ is active. The converse is false (an active shell may dissolve before *shell_search* is executed on it).

To define the third type, say that edge e *crosses* a set of vertices B if one end is in B, i.e., $|e \cap B| = 1$. (In this notation B is usually a blossom.) The *crossing function* $\gamma : 2^{V(G)} \rightarrow \mathbf{Z}$ of a matching M is defined by $\gamma(B) = |\{e | e \in M \text{ crosses } B\}|$. For example if B is a blossom of the matching, $\gamma(B) \leq 1$. The third type of shell $G(C, D)$ is *uncrossed* if the current matching does not cross C or D, i.e., $\gamma(C) = \gamma(D) = 0$. (For an odd shell this amounts to $\gamma(C) = 0$.) An undissolved shell is certainly uncrossed, but the converse is false.

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 that is similar to the witness inequality but unfortunately has some extra terms.

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 blossom of $P(R)$ possible. 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 $\gamma_0(B) = (\textbf{if } \omega \in B \textbf{ then } 0 \textbf{ else } 1)$. (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 $P(R)$ (C or D may be currently dissolved or undissolved). Let F_ω be the set of free vertices of $G(C,D) - \omega$. (Possibly F_ω is empty.) 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 (γ, $\widehat{\delta}$, F_ω) are evaluated at the chosen time in the execution of *path*(R). (The lemma does not hold after the execution of *path*(R), as indicated in the proof.)

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

$$(\gamma - \gamma_0)\widehat{\delta}(\,(C,D) \cup \mathcal{DN}[C,D)\,) + \widehat{\delta}(F_\omega,\ V(T_0)) \le 5\widehat{n}(C,D).$$

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

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

In the last definition B is a blossom, old or current. We say that an old blossom "intersects" a shell if they have a vertex (or edge) in common. Thus the old blossoms that intersect $G(C,D)$ are those in $[G^*,D) \cup \mathcal{DN}[C,D)$. 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 $\widehat{n}(C)$ is odd, $\omega \in C$. Hence M_0 does not cross C. Similarly for D, if it is nonempty.

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

$$\widehat{y_0 z_0}(e) \le c(e), \qquad \text{for } e \in M;$$
$$\widehat{y_0 z_0}(e) \ge c(e) - 8, \qquad \text{for } e \in M_0.$$

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

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

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

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

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

Again this 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 to bound the $V(T_0)$ terms:

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

First we prove the equation $2\mu(B) = (f + \gamma' - \gamma_0')(B)$: Let v be the number of vertices in $V(B) \cap G(C, D)$. By definition $2\mu(B) = (v - 2|M' \cap G(B)|) - (v - 2|M_0' \cap G(B)|)$. The right-hand side is how many more vertices of $V(B) \cap G(C, D)$ M does not match on edges of $G(B) \cap G(C, D)$ than M_0. 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 $V(B) \cap G(C, D)$. Vertices of the first type contribute $f(B)$ to the right-hand side (note that ω is free in both matchings). Vertices of the second type contribute $(\gamma' - \gamma_0')(B)$, since no edge of either matching crosses D. This gives the desired equation.

It is easy to see that to prove the inequality for the $V(T_0)$ terms, it suffices to show that an old blossom B has $(z_0 - z)(B) \geq 2\widehat{\delta}(B)$ or $\mu(B) = 0$ (here we use the nonnegativity of $\mu(B)$). If B has never become a current blossom then $(z_0 - z)(B) = 2\widehat{\delta}(B)$. If B is a current blossom then $\mu(B) = 0$. If B dissolved by becoming a blossom but is not a current blossom then $(z_0 - z)(B) = z_0(B) \geq 2\widehat{\delta}(B)$. The inequality for $V(T_0)$ terms follows.

Next we deduce

$$(y - y_0)(F_\omega) + (f + \gamma' - \gamma_0')\widehat{\delta}(\ [R, D) \cup \mathcal{DN}[C, D)\ ) \ \le\ 5\widehat{n}(C, D).$$

This follows by adding the two d estimates and replacing the $\mu(z_0 - z)$ terms using the above observations. In addition note that $|M'| \le |M_0'| \le \widehat{n}(C, D)/2$; also, $\widehat{\delta}$ vanishes on $[G^*, R)$, since we have chosen a time during the execution of $path(R)$.

A free vertex v has $y(v) = y_0(v) - (\widehat{\delta} - \widehat{\delta}(v, \ \cdot\ ))[G^*, v)$. (To show this consider any execution *shell_search*(A, B). A dual adjustment of δ does not change $y(v)$ if $v \notin A$ or $v \in G(A, B)$; it decreases $y(v)$ by 2δ if $v \in B$ (which implies $v \in A$).) Summing these equations for all $v \in F_\omega$ implies

$$(y - y_0)(F_\omega) + f\widehat{\delta}(\ [R, D) \cup \mathcal{DN}[C, D)\ ) - \widehat{\delta}(F_\omega, V(T_0)) = 0.$$

The last step is to subtract the last equation from the preceding inequality. This gives the lemma, if we use two observations: The terms $((\gamma' - \gamma_0')\widehat{\delta}([R, C])$ vanish, since a blossom $B \in [R, C]$ has $V(B) \cap V(C) = V(C)$ and $\gamma'(C) = \gamma_0'(C) = 0$. The remaining blossoms B are contained in $G(C, D)$. Hence the function $\gamma' - \gamma_0'$ simplifies to $\gamma - \gamma_0$. ∎

The rest of the analysis involves two quantities Δ, ϵ. Before defining them we give some motivation, and in the process we survey the rest of the derivation. Observe that every scale 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. In bipartite matching this is the only source of error [GT87]. In general matching there is a second type of error when $path(R)$ begins. It comes from changes in the duals made by calls $path(C)$, for children C of $P(R)$. Specifically in the inequality of Lemma 6.1, the right-hand side corresponds to the error caused by scaling. If the first term on the left-hand side is nonnegative, Lemma 6.1 is essentially the desired witness inequality (it is even stronger). However an uncrossed blossom $B_0 \in \mathcal{DN}[C, D)$ that does not contain ω makes a negative contribution to the first term on the left. This is the second type of error, coming from dual adjustments in previous calls to *path*. (It is tempting to conjecture that the negative contribution of B_0 is offset by terms $\widehat{\delta}(v, B_0)$ in the second term. In general this is false: An execution *shell_search*(B_0, C_0) can contribute to the first term because it translates B_0, but not contribute to the second term because $G(B_0, C_0)$ does not contain any vertex that is currently free.)

The quantity $\epsilon(C)$ measures the amount of error introduced by $path(C)$ (including calls for all descendants of C). Lemma 6.4 below shows that processing a major path $P(R)$ adds $O(\widehat{n}(R))$

to the error, i.e., the total error introduced by *path*(R) (including calls for its descendants) is $O(\widehat{n}(R)\log\widehat{n}(R))$. The desired witness inequality follows quickly from this bound.

To bound ϵ we use the quantity Δ, which measures total dual adjustment. Lemma 6.3 below shows that ϵ is bounded by a sum involving Δ terms. Lemma 6.2 below shows that Δ is itself bounded by a sum involving Δ terms. These two results plus the above Lemma 6.1 combine to give the desired Lemma 6.4.

Here are the formal definitions. For a shell $G(C, D)$ of $P(R)$,

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

Equivalently $\Delta(C, D)$ is twice the total of all dual adjustments made in searches of even shells $G(A, B)$ $(A, B \in [C, D])$, plus if $D = \emptyset$ the dual adjustments of odd shells $G(A)$ $(A \in [C, D])$. For a major path root R the odd shell R has $\Delta(R) = \widehat{\delta}(P(R))$. 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 have a term corresponding to this dual adjustment whereas the alternative definition does.)

Unlike Δ, ϵ is a time-varying quantity. To define it fix a time in the algorithm, and let γ be the crossing function and F the set of free vertices, both defined for the current matching. For an old blossom C the quantity

$$(\gamma\widehat{\delta} + \widehat{\delta}(F, \cdot\,))(C)$$

represents the total amount of translations of C that have been "witnessed" by either a currently free vertex or a currently matched edge crossing C (an edge crossing C "witnesses" every translation of C, unlike a free vertex). For an old blossom B, $\epsilon(B)$ is the total of all "unwitnessed" translations of blossoms contained in B, more precisely,

$$\epsilon(B) = ((1 - \gamma)\widehat{\delta} - \widehat{\delta}(F, \cdot\,))(\mathcal{D}(B)).$$

This quantity changes because (i) $\widehat{\delta}$ increases as more translations of blossoms are done; (ii) F gets smaller as more vertices are matched; (iii) γ changes as the matching changes. To see how this definition fits into the above motivation, first observe that Lemma 6.1 can be rewritten as follows.

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

$$(\gamma-\gamma_0)\widehat{\delta}(\ (C,D)\ )+\widehat{\delta}(F_\omega,P(R))\leq 5\widehat{n}(C,D)+\epsilon(\ \mathcal{N}[C,D]\ )$$

where all quantities (including ϵ) are evaluated at the chosen time in *path*(R).

Proof. This follows by rearranging the inequality of the lemma. Note that in that inequality, "$V(T_0)$" is equivalent to $P(R)\cup\mathcal{DN}[C,D]$. To get the ϵ terms on the right-hand side use two facts about old blossoms B: If $\omega\notin B$ then $\gamma_0(B)=1$. If $\omega\in B$ then $\gamma_0(B)=0$ and $(\widehat{\delta}-\widehat{\delta}(\omega,\ \cdot\ ))(B)\geq 0$. ∎

The corollary indicates that a bound on $\epsilon(B)$ for $B\in\mathcal{N}[C,D]$ can be used to bound the total dual adjustment, and complete the analysis (we shall see that the first term on the left-hand side is nonnegative). It is important to note, though, that the bound on $\epsilon(B)$ must hold even *after* the call to *path*(B) (thus see the statement of Lemma 6.4). In evaluating $\epsilon(B)$ after *path*(B), in the definition of $\epsilon(B)$ the $\widehat{\delta}$ functions count the total of all translations ever made, and "free" and γ refer to the *current* matching.

Now we start our program of bounding first Δ, then ϵ. We often use an interval $[A,B)$ in the blossom tree to refer to the shell $G(A,B)$. The interval is *active, uncrossed*, etc. if the corresponding shell has that property.

The argument works by partitioning various intervals (shells) into subintervals (subshells). Observe that the active intervals are nested: If $[A,B)$ is active and $C\in(A,B)$ then any active interval having C as a boundary is contained in (A,B) (i.e., its other boundary is in $(A,B]$). Thus any even shell $[A,B)$ of $P(R)$ can be partitioned into active shells $[A_i,A_{i+1})$. Specifically $A_1=A$, and A_{i+1} is defined so that $[A_i,A_{i+1})$ is the largest possible active shell contained in $[A,B)$ with outer boundary A_i. (A_{i+1} exists since any even original shell is active.) We use this partition in Lemmas 6.2–6.3.

Now we estimate Δ for an active shell $G(C,D)$ of $P(R)$. Throughout the following lemma and proof, "shell" refers to a shell of $P(R)$. The calculations rely on the fact that any shell (of $P(R)$) contained in $G(C,D)$ is even. Out of all dual adjustments made for shells contained in $G(C,D)$ consider the last one. (This adjustment can be for shell $G(C,D)$ or a smaller shell; in the latter case, a subsequent dual adjustment is made for a shell with outer boundary D (or inner boundary C) that dissolves D (C).) Let M be the matching at the time of this last dual adjustment and let γ be the crossing function of M.

Lemma 6.2. An active interval $[C, D)$ of $P(R)$ can be partitioned into an uncrossed interval $[T, U)$ and a set of active intervals $\mathcal{A}$ such that for F the set of free vertices of M in $[T, U)$,

$$\Delta(C, D) \leq \gamma\widehat{\delta}([T, U)) + \widehat{\delta}(F, [C, U)) + \Delta(\mathcal{A}).$$

Proof. To define the partition, choose any free vertex v in $[C, D)$ and let $[T, U)$ be the minimal uncrossed shell containing v. (Thus T is the innermost blossom of $P(R)$ containing v with $\gamma(T) = 0$ and U is the outermost blossom of $P(R)$ not containing v with $\gamma(U) = 0$.) Clearly $[T, U) \subseteq [C, D)$, since $\gamma(C) = \gamma(D) = 0$. Let $\mathcal{A}$ consist of the above partition of each of the even shells $[C, T)$, $[U, D)$ into active intervals. (One or both of these even shells may be empty.)

To verify the inequality of the lemma, consider any shell $G(A, B) \subseteq G(C, D)$ that gets searched. Let Δ_0 denote the total of all translations made in all searches of shell $G(A, B)$. We show that Δ_0 is counted by the terms on the right-hand side of the inequality. "Counted" means that the terms contain a contribution of Δ_0 uniquely associated with the translations for $G(A, B)$.

This is obvious if $[A, B)$ is included in one of the active intervals of $\mathcal{A}$. There are two other cases: (*i*) $A \in (T, U)$; (*ii*) A is the boundary of a shell of $\mathcal{A}$ and B is a proper subset of T. This follows from the nesting property of active intervals.

First consider case (*i*). If $B \in (T, U)$ then Δ_0 is counted by $\gamma\widehat{\delta}(\{A, B\})$, since $\gamma(A), \gamma(B) \geq 1$. Suppose $B \notin (T, U)$. Thus $[A, U) \subseteq [A, B)$. Since $[A, U)$ is even and U is uncrossed, either $\gamma(A) \geq 2$ or $\gamma(A) = 1$ and $G(A, U)$ contains a vertex of F. In either case Δ_0 is counted by $(\gamma\widehat{\delta} + \widehat{\delta}(F, \cdot\,))(A)$.

Next consider case (*ii*). If $B \subseteq U$ then $[T, U) \subseteq [A, B)$. Since $G(T, U)$ has at least two free vertices, Δ_0 is counted by $\widehat{\delta}(F, A)$. The other possibility is $B \in (T, U)$. Thus $[T, B) \subseteq [A, B)$. As above, either $\gamma(B) \geq 2$ or $\gamma(B) = 1$ and $G(T, B)$ contains a vertex of F. Thus Δ_0 is counted by $\gamma\widehat{\delta}(B) + \widehat{\delta}(F, A)$. ∎

Corollary 6.2. For an active shell $G(C, D)$ of $P(R)$,

$$\Delta(C, D) \leq 5\widehat{n}(C, D) + \epsilon(\,\mathcal{N}[C, D)\,),$$

where each quantity $\epsilon(B), B \in \mathcal{N}[C, D)$, is evaluated at a time in *path*(R) (the times in *path*(R) may differ).

Proof. The proof is by induction on $\widehat{n}(C, D)$. Corollary 6.1 implies that the uncrossed interval $[T, U)$ of the lemma has $\gamma\widehat{\delta}(\,[T, U)\,) + \widehat{\delta}(F, [C, U)) \leq 5\widehat{n}(T, U) + \epsilon(\,\mathcal{N}[T, U)\,)$, where all quantities (including ϵ) are evaluated after the last dual adjustment of a shell in $G(C, D)$. (Note that since

$[C,D)$ is active and even, $\omega \in D$ at this time, whence γ_0 vanishes on $[T,U)$. Also since T is uncrossed, the argument to $\gamma\widehat{\delta}$ is correct. Finally since $[C,D)$ is active, the second argument to $\widehat{\delta}$ is correct.) Substituting in the lemma and applying the inductive hypothesis to each interval of $\mathcal{A}$ gives

$$\Delta(C,D) \le 5\widehat{n}(T,U) + \epsilon(\,\mathcal{N}[T,U)) + 5\widehat{n}(\mathcal{A}) + \epsilon(\mathcal{N}(\mathcal{A})).$$

This implies the corollary. ▮

Now we estimate $\Delta(R)$ for a major path root R. Throughout the following lemma and proof, "shell" refers to a shell of $P(R)$. Let M be a matching with free vertices F. Let γ be the crossing function of M. We allow M to cross R, i.e., possibly $\gamma(R) > 0$; this can occur in executions of *path*(R') for R' an ancestor of R. (Because of this the estimate is done slightly different than Lemma 6.2. To see why first note that Lemma 6.2 gets used in the form of Corollary 6.2; the latter depends on Corollary 6.1, which in turn depends on Lemma 6.1; but Lemma 6.1 is valid only during the execution of *path*(R).)

Lemma 6.3. For some (possibly empty) blossom D in $P(R)$, interval $[R,D)$ can be partitioned into the original shells (intervals) of $P(R)$ containing a vertex of F, plus a set of active intervals $\mathcal{A}$, such that for $\widehat{\delta}$ evaluated at the end of *path*(R),

$$\Delta(R) \le (\gamma\widehat{\delta} + \widehat{\delta}(F,\ \cdot\ ))(P(R)) + \Delta(\mathcal{A}).$$

Proof. The argument has the same form as Lemma 6.2. Let $[C,D)$ be the innermost original shell of $P(R)$ that contains a vertex of F (possibly C is the innermost nonleaf blossom of $P(R)$ and $D = \emptyset$). The portion of $[R,D)$ that excludes the original shells containing a vertex of F consists of a number of even shells. Let $\mathcal{A}$ consist of the above partition of each of these even shells into active intervals.

To verify the inequality of the lemma consider any shell $G(A,B)$ of $P(R)$ that gets searched. Let Δ_0 denote the total of all translations made in all searches of shell $G(A,B)$ (if $B = \emptyset$, Δ_0 denotes the total translation of A). As in Lemma 6.2 we show that Δ_0 is counted by the terms on the right-hand side of the inequality.

This is obvious if $[A,B)$ is included in one of the active intervals of $\mathcal{A}$. There are two other cases: (i) $A \subseteq D$; (ii) A is the boundary of a shell of $\mathcal{A}$ and $B \subseteq U$, where $[T,U)$ is the outermost original shell containing a vertex of F and contained in A. This follows from the nesting property of active intervals.

First consider case (i). Clearly D is nonempty in this case, so it has an odd number of vertices but no free vertex of M. Hence the same applies to any blossom $X \subseteq D$, whence $\gamma(X) \geq 1$. Thus Δ_0 is counted by $\gamma\widehat{\delta}(A)$ if $B = \emptyset$ and $\gamma\widehat{\delta}(\{A, B\})$ if $B \neq \emptyset$.

Next consider case (ii), where we have $[T, U) \subseteq [A, B)$. Since $G(T, U)$ has a free vertex, if $B = \emptyset$ then Δ_0 is counted by $\widehat{\delta}(F, A)$.

Suppose $B \neq \emptyset$. Since $[A, B)$ is even and contains a vertex of F, either $\gamma(\{A, B\}) \geq 1$ or $\gamma(A) = \gamma(B) = 0$ and $[A, B)$ contains two vertices of F. In either case Δ_0 is counted by $\gamma\widehat{\delta}(\{A, B\}) + \widehat{\delta}(F, A)$. ▮

Lemma 6.4. For a major path root R at any time, even after $path(R)$, $\epsilon(R) \leq 5\widehat{n}(R)\lfloor \log \widehat{n}(R) \rfloor$.

Proof. Let the rank of R be $r = \lfloor \log \widehat{n}(R) \rfloor$. The argument is by induction on r. The base case is $r = 0$, i.e., R is a leaf blossom. Such a blossom is never translated, so $\epsilon(R) = 0$ and the desired inequality holds.

For the inductive step let R have rank $r > 0$; assume the lemma for roots of rank less than r and prove it for R as follows. For the current matching M, let F be the set of free vertices and let γ be the crossing function. M need not be contained in R, i.e., possibly $\gamma(R) > 0$. Using the definition of $\epsilon(R)$ and then the partition set $\mathcal{A}$ of Lemma 6.3 gives

$$\epsilon(R) = (\epsilon(\mathcal{N}(\,\cdot\,)) + (1-\gamma)\widehat{\delta} - \widehat{\delta}(F, \,\cdot\,))(P(R)) \leq \epsilon(\mathcal{N}(P(R))) + \Delta(\mathcal{A}).$$

Now we bound the two terms on the rightmost side.

For the ϵ term, consider any $B \in \mathcal{N}(P(R))$. The inductive hypothesis shows that $\epsilon(B) \leq 5(r-1)\widehat{n}(B)$. Furthermore an old blossom B with $V(B) \cap F = \emptyset$ has $\gamma(B) \geq 1$; this implies that such a blossom has $\epsilon(B) \leq 0$. Thus the first term is bounded by

$$\epsilon(\mathcal{N}(P(R))) \leq 5(r-1)\widehat{n}(\{B \mid B \in \mathcal{N}(P(R)), V(B) \cap F \neq \emptyset\}).$$

For the Δ term, use Corollary 6.2 and then the inductive hypothesis for each nonmajor child of an interval of $\mathcal{A}$, to get

$$\Delta(\mathcal{A}) \leq 5\widehat{n}(\mathcal{A}) + \epsilon(\mathcal{N}(\mathcal{A})) \leq 5\widehat{n}(\mathcal{A}) + 5(r-1)\widehat{n}(\mathcal{N}(\mathcal{A})) \leq 5r\widehat{n}(\mathcal{A}).$$

Using the bounds for the two terms gives the desired inequality,

$$\epsilon(R) \leq 5(r-1)\widehat{n}(\{B \mid B \in \mathcal{N}(P(R)), V(B) \cap F \neq \emptyset\}) + 5r\widehat{n}(\mathcal{A}) \leq 5r\widehat{n}(R).$$

Here we have used the fact that no shell of $\mathcal{A}$ contains a vertex of F. ∎

Let R be a major path root. At any time in *path*(R) let F be the set of free vertices of R, and let ω be a free vertex in the innermost blossom of R possible. The choice of time in *path*(R) implies that $\gamma(R) = 0$ and ω exists. Hence any vertex $v \in F - \omega$ is in a minimal uncrossed shell $G(A, B)$ of $P(R)$ (A is the innermost blossom of $P(R)$ containing v with $\gamma(A) = 0$; B is the outermost blossom of $P(R)$ not containing v with $\gamma(B) = 0$, or if such does not exist $B = \emptyset$). Let $\mathcal{U}$ be the set of intervals $[A, B)$ for all such shells. Apply Corollary 6.1 to each shell of $\mathcal{U}$. Note that a blossom $B \in P(R)$ has $\gamma(B) \geq \gamma_0(B)$ (since $\gamma_0(B) =$ (**if** $\omega \in B$ **then** 0 **else** 1), and $\omega \notin B$ implies $\gamma(B) > 0$). Thus

$$\widehat{\delta}(F - \omega, P(R)) \leq 5\widehat{n}(R) + \epsilon(\mathcal{N}(\mathcal{U})) \leq 5\widehat{n}(R) \log \widehat{n}(R),$$

where the last inequality follows from Lemma 6.4 and the fact that the intervals of $\mathcal{U}$ are disjoint. This is the desired witness inequality.

Next we derive an inequality used to implement the priority queue in *shell_search* (see Section 7).

Corollary 6.3. For a major path root R, the total dual adjustment in all phase 2 *shell_searches* of *path*(R) is at most $5\widehat{n}(R) \log \widehat{n}(R)$.

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, i.e., adjustment when all free vertices in R' are in the odd shell. We show that each d_i is at most $(5/2)\widehat{n}(R) \log \widehat{n}(R)$.

The dual adjustments counted in d_1 all occur in *shell_search* of an even shell. Corollary 6.2 and Lemma 6.4 show that for an active shell $G(C, D)$ of $P(R)$, $\Delta(C, D) \leq 5\widehat{n}(C, D) \lfloor \log \widehat{n}(C, D) \rfloor$. Summing these inequalities for all maximal active shells of $P(R)$ and using the definition of Δ implies that $d_1 \leq (5/2)\widehat{n}(R) \log \widehat{n}(R)$.

For d_2, consider the uncrossed shell $G(R, \emptyset)$ 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, i.e., $\widehat{\delta}(v, P(R)) \geq d_2$. Thus the witness inequality implies $d_2 \leq (5/2)\widehat{n}(R) \log \widehat{n}(R)$. ∎

In the proof the bound used for d_1 actually bounds the total dual adjustment in all *shell_searches* of even shells (in phases 1 or 2). Hence this total is at most $(5/2)\widehat{n}(R) \log \widehat{n}(R)$. This fact is also used in Section 9.

7. The Search Step.

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

At the start of the *match* routine, the old blossom tree T_0 is ordered so 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 $\{vw|w < v\}$, ordered by decreasing w. The other list contains the edges $\{vw|w > v\}$, 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 (even or odd) $[C, D)$ constitute the interval $[lo(C)..lo(D))$. Hence for any vertex v in an undissolved shell $[C, D)$, 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 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 the previous section, it is convenient to let $n = \widehat{n}(R)$ and $m = \widehat{m}(R)$. The undissolved blossoms of $P(R)$ are stored in a doubly-linked list $\mathcal{U}$; the order of blossoms in $\mathcal{U}$ is the same as in $P(R)$.

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

Now consider the Search Step. First observe the *disjointness property*: In one Search Step, a given vertex is involved in at most one execution of *shell_search*, and a given edge is examined in at most one execution of *shell_search*. 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 relevant only 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 $\mathcal{U}$, i.e., the smallest undissolved blossom containing C. C is deleted from $\mathcal{U}$. 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 $[lo(B), lo(D))$). The disjointness property implies that the total time for scanning edges in dissolve steps in one Search Step is $O(m)$. (Some additional processing done when a blossom dissolves, concerning dual values, is discussed below.)

Next consider the time in the Search Step associated with the priority queue used 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 is made. Blossom steps are implemented in linear time using the incremental tree set-merging algorithm of [GT85]. This makes the time for one Search Step in phase 1 $O(m)$.

The total time for phase 3 is $O(m \log n)$. This can be achieved by implementing the priority queue as a balanced tree [GMG]. [GGS] gives an even better bound, but this is not needed here.

For phase 2, Corollary 6.3 shows that the total dual adjustment is at most $5n \log n$. Hence the minimum computations for dual adjustments can be done using a queue of $5n$ entries, plus *log n* lists of entries for future queues. All shells share the same queue, to avoid reinitialization. This makes the total overhead for the queue in all phase 2 searches $O(n \log n)$ (by the disjointness property). (See [GT87] for a more detailed discussion of the implementation of such a queue.) To implement *expand* steps the list-splitting algorithm of [G85b] is used. This makes the time for one Search Step in phase 2 $O(m\alpha(m,n))$ (by the disjointness property).

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 [GMG, GGS]; 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. We also 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 $\widehat{yz}(e)$ in $O(1)$ time. These two, plus the details in [GMG, GGS], give the desired time bound for our algorithm. Recall the interval notation introduced in Section 4, e.g., $[G^*, B] = \{A \mid A \text{ is an ancestor of } B \text{ in } T_0\}$, and $z[G^*, B]$ denotes $z([G^*, B])$.

We start by describing the data structure. The algorithm stores two values for each old blossom B, $z_1(B)$ and $t(B)$. During the computation z_1 keeps track of sums of z_0 values and t keeps track of total amounts of translations; initially $z_1(B) = z_0[G^*, B]$ and $t(B) = 0$. At any time the true value of $y(v)$ differs from the value that the algorithm stores (in an array) as $y(v)$; call the latter $y'(v)$. The algorithm maintains these two invariants:

$$y(v) = y'(v) - t(\{A \mid A \text{ an undissolved ancestor of } v \text{ in } T_0\}),$$

$$z[G^*, B] = z_1(B) - 2t(\{A \mid A \text{ an undissolved ancestor of } B \text{ in } T_0\}).$$

Consider a search of shell $G(C, D)$. To calculate $\widehat{yz}(e)$, write $\widehat{yz}(e) = y(e) - z[G^*, C] - z\{B \mid B \text{ is a current blossom containing } v\}$. The last term is calculated as in Edmonds' algorithm, so we concentrate on $y(e) - z[G^*, C]$. This equals $y'(e) - z_1(C)$. Hence $\widehat{yz}(e)$ can be calculated in time $O(1)$ 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 decreasing each of $z(C)$ and $z(D)$ by 2δ and increasing $t(C)$ and $t(D)$ by δ. This has the same effect as a translation, and maintains the above two invariants. Hence the dual adjustment is done correctly in time $O(1)$.

Next suppose that blossom C or D dissolves in the *shell_search.* We first describe the case of C dissolving. Let B be the smallest undissolved blossom containing C. The algorithm assigns $z_1(B) \leftarrow z_1(C)$, $t(B) \leftarrow t(C)$, and for each vertex $v \in B - C$, $y'(v) \leftarrow y'(v) + t(C) - t(B)$. Since C has dissolved, $z[G^*, C] = z[G^*, B]$. Thus it is easy to see the invariants are maintained. The disjointness property implies that in one Search Step the total time spent reassigning y' values is $O(n)$. The rest of the time is $O(1)$ per dissolve step, as desired. The case of D dissolving is similar, but only y' values get changed.

A degenerate case of this is the end of the scale, when G^* dissolves by becoming current. At that time the true y values are computed from y' and t, as above (there is no B blossom).

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 $\mathcal{P}$ of vertex-disjoint augmenting paths. We present an algorithm based on depth-first search and the properties of blossoms [E65a].

The algorithm grows a search graph $\mathcal{S}$. The structure of $\mathcal{S}$ is the similar to Edmonds' weighted matching algorithm (Section 1.1) except for three changes: First, the requirement that an edge of $\mathcal{S}$ be tight is dropped (there are no edge costs). Second, every inner blossom is a vertex, not a nonleaf blossom. (This comes about because the routine 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 $\mathcal{S}$ one at a time. A free vertex v is either outer (if some search starts from v) or inner (if a search ends by finding an augmenting path to v). The contracted subgraph $\mathcal{F}$ (of Section 1.1) is always a forest: an augmenting path corresponds to a path in $\mathcal{S}$ joining an outer free vertex to an inner free vertex.

The final difference in $\mathcal{S}$ from Edmonds' algorithm is that the search is done depth-first. Recall that in an ordinary depth-first search of a directed or undirected graph, the search path leading to the vertex currently being scanned contains all vertices that have not been completely scanned [AHU]. Our depth-first search has this property (property (vi) below) which is crucial to its efficiency.

Recall that each outer vertex x of $\mathcal{S}$ has an even length alternating path from x to a free vertex; we denote this path $P(x)$. We now describe the data structure that specifies these paths. 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 P, Q 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.

Let F be the set of free vertices of M. For a vertex $y \notin F$, y' denotes the vertex matched to y (in M). Each outer vertex x has a label $\ell(x)$ that defines path $P(x)$, as follows ([G76]). A label is either a *singleton label* $\ell(x) = y$, where y is an outer vertex, or a *pair label* $\ell(x) = (y,z)$, where y and z are outer vertices and the pair is ordered. If x has a singleton label $\ell(x) = y$ then $P(x) = x, x', P(y)$. (A degenerate case is a free vertex x, which has $\ell(x) = \emptyset$ and $P(x) = x$.) If x has a pair label $\ell(x) = (y,z)$ then $x \in P(y)$ and $P(x) = P(y,x)^r P(z)$.

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

The algorithm consists of a main routine *find_ap_set* and a recursive procedure *find_ap*. *find_ap_set* initializes the search graph $\mathcal{S}$ to empty and each $b(v)$ to v. Then it examines each vertex $x \in F$ in turn. If x is not in a path of $\mathcal{P}$ when it is examined, the routine adds x to $\mathcal{S}$ (by assigning $\ell(x) \leftarrow \emptyset$) and calls the recursive procedure *find_ap*(x), stated in pseudo-Algol below. Procedure *find_ap* either ends normally or gets terminated before normal completion, by a recursive call. (The latter occurs when a recursive call discovers an augmenting path.) A vertex x is designated *scanned* if the invocation *find_ap*(x) has ended normally.

```
procedure find_ap($x$)  {$x$ is an outer vertex}
for each edge $xy \notin M$ do  {examine an edge}
    if $y \notin V(\mathcal{S})$ then
        if $y$ is free then begin  {an augmenting path has been found}
            add $xy$ to $\mathcal{S}$, and add path $yP(x)$ to $\mathcal{P}$
            terminate all active recursive calls to find_ap
            end
        else begin  {grow steps}
            add $xy, yy'$ to $\mathcal{S}$, by setting $\ell(y') \leftarrow x$
            $find\_ap(y')$
            end
    else if $y$ is a scanned outer vertex and $b(y) \neq b(x)$ then begin  {blossom step}
        let $u_i,\ i = 1, \ldots, k$ be the inner vertices in $P(y, b(x))$, ordered so that $u_i$ precedes $u_{i-1}$
        for $i \leftarrow 1$ to $k$ do begin  {update $\mathcal{S}$}
            $\ell(u_i) \leftarrow (y, x)$
            for each vertex $v$ with $b(v) \in \{u_i, u_i'\}$ do $b(v) \leftarrow b(x)$
            end
        for $i \leftarrow 1$ to $k$ do $find\_ap(u_i)$
        end  ▮
```

Figure 8.1 shows the search graph constructed by $find\_ap(1)$. Vertices are labelled in the order they become outer.

Now we show that the algorithm is correct and *find_ap_set* halts with a maximal set of augmenting paths. In the discussion it is convenient to let f denote the free vertex that is the current root of $\mathcal{F}$.

The above algorithm uses several high-level concepts, for clarity and flexibility for a detailed implementation. We now give the low-level definitions of these concepts needed to prove correctness. Call vertex x "outer" if it has received a label $\ell(x)$. Call x "inner" if it is not outer but its mate x' is. The search graph $\mathcal{S}$ contains all outer and inner vertices (we need not specify $E(\mathcal{S})$ to implement the algorithm). Note that the proof of correctness below shows that the terms "outer", "inner" and $\mathcal{S}$ correspond exactly to their definitions in Edmonds' algorithm.

We adopt one more convention, to ensure that the algorithm is well-defined: In the first line of the blossom step, it is conceivable that $P(y, b(x))$ is undefined because $b(x) \notin P(y)$. In this case interpret $P(y, b(x))$ as $P(y)$; also in updating $\mathcal{S}$, assign $b(v) \leftarrow f$. (Property (*vii*) below shows that this case never occurs.)

It is convenient to refer to the recursion forest R for *find_ap*. More precisely R is a forest whose roots are the free vertices g such that *find_ap_set* calls $find\_ap(g)$. In addition if $find\_ap(x)$ calls $find\_ap(y)$ then x is the parent of y. Figure 8.2 shows the recursion tree for Figure 8.1.

An important aspect of correctness is that the augmenting paths found by the algorithm are simple. (It is well-known that augmenting a matching along a nonsimple augmenting path can give a set that is not a matching.) This amounts to showing that any path $P(x)$ is simple. We accomplish this only at the end of the development (Lemma 8.1). For convenience define a *walk* [H] to be path that need not be simple. Terms defined for paths have the obvious meaning for walks.

The proof of correctness begins with eight basic properties of *find_ap*, labelled (i) – $(viii)$. Each property has a simple proof, usually by induction on the number of steps of the algorithm. We omit most of the details of the proofs which are straightforward, but give the most important points. (Further details can be found in [G76]; our algorithm is essentially a special case of [G76], using a depth-first rule for edge selection).

(i) For any outer vertex x, $P(x)$ is an even length alternating walk from x to f.

This is immediate, except for the fact that $P(x)$ is even when x has a pair label. The latter follows from the observation that in any path $P(z)$, an inner vertex is an odd distance from f; this observation, in turn, follows by an easy induction.

(ii) If z is a proper ancestor of x in R then $P(x) = P(x, z_-)P(z)$, for z_- the vertex preceding z in $P(x)$.

(iii) For any outer vertex x, if u is an inner vertex in $P(x)$ then u' is an ancestor of x in R.

Note that (ii) – (iii) combine to show that if u is an inner vertex in $P(x)$ with $u' \neq x$ then $P(x) = P(x, u'_-)P(u')$. We shall use this combination of (ii) – (iii) several times.

(iv) At any time $b(x)$ is the first vertex in $P(x)$ with $b(x)'$ inner.

To make property (iv) true interpret f' as an inner vertex. Note that $b(x)$ changes as the algorithm progresses. The proof uses (ii) – (iii), and also our convention for blossom steps when $b(x) \notin P(y)$.

(v) For any outer vertex x, every vertex $v \in P(x, b(x))$ has $b(v) = b(x)$.

This proof uses (ii) – (iii).

For the next property say that *find_ap* "examines an edge" each time control passes to the line so-labelled (i.e., the first line). This includes the last time, when no more edges xy exist.

(vi) Each time $find\_ap(x)$ examines an edge, every unscanned outer vertex is in $P(x) \cup \mathcal{P}$.

This property follows from the order in which a blossom step calls $find\_ap(u_i)$.

(vii) In a blossom step, $b(x) \in P(y)$.

The argument refers to two times in the algorithm: let t_y be the time when immediately after $find\_ap(y)$ has processed edge xy; let t_x be the time when $find\_ap(x)$ does a blossom step for xy. For emphasis we write $b_x(z)$ to denote the value of $b(z)$ at time t_x. Thus property (vii) refers to $b_x(x)$. We will show that $b_x(x)$ is an unscanned outer vertex at time t_y. This gives the desired conclusion $b_x(x) \in P(y)$, by property (vi) applied to $find\_ap(y)$.

At time t_y, x and x' are in $\mathcal{S}$. Let z be x if x is outer at time t_y, else x'. Note that $b_x(z) = b_x(x)$. Since $b_x(x) \in P(z)$, $b_x(x)$ is in $\mathcal{S}$ at time t_y. By (iv), $b_x(x)'$ is inner at time t_x, hence it is inner at time t_y. By (iii) $b_x(x)$ is an ancestor of x. Since x is unscanned at t_y, $b_x(x)$ is an unscanned outer vertex at t_y, as desired.

$(viii)$ For any edge xy with x and y scanned, $b(y) = b(x)$.

Let x be scanned after y. We can assume that $b(x) \neq b(y)$ when $find\_ap(x)$ examines edge xy. Hence a blossom step is executed for xy. It sets $b(y)$ to $b(x)$ if $k \geq 1$ (see $find\_ap$). The latter follows from $b(x) \neq b(y)$ and property (vii).

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

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

Proof. To show that each path of $\mathcal{P}$ is augmenting we need only show that each path $P(x)$ is simple. This is done by induction. In the inductive step the case that x has a singleton label is clear. So consider a vertex u_i that has a pair label (y, x). Suppose $P(u_i)$ contains a vertex v more than once. By induction v occurs in both paths $P(y, u_i)$ and $P(x)$. We show this cannot be, by showing $v \notin P(x, b(x))$ and $v \notin P(b(x))$. Since $b(v) = u'_j$ for some $j \geq i$, $v \notin P(x, b(x))$, by (v). Since $P(y)$ is simple, $v \notin P(b(x))$. ($P(y)$ contains $P(b(x)$ by property (vii) and properties (ii) – (iv) combined.) Thus $P(u_i)$ is simple, as desired.

It is clear that the paths of $\mathcal{P}$ are disjoint. Now we show that when *find_ap_set* halts, $\mathcal{P}$ is maximal, i.e., 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 is vertex-disjoint from $\mathcal{P}$ and starts at a free vertex x_0. (Note that the vertices x_i need not all be in the same search tree of *find_ap*.) Observe that when *find_ap_set* halts every outer vertex $x \notin \mathcal{P}$ has been scanned, by (vi). Hence any edge xy with $y \notin \mathcal{P}$ has y matched and either inner or outer; in the latter case $b(x) = b(y)$, by (iii). Now we show by induction that every x_{2j} is outer and $b(x_{2j}) = x_h$ for some $h \leq 2j$. (Note that this implies the alternating path is not augmenting, as desired.)

The base case $j = 0$ is obvious. For the inductive step assume that x_{2j} is outer. If x_{2j+1} is inner then it is matched (recall an outer vertex is not adjacent to a free vertex); further x_{2j+2} is outer, and $b(x_{2j+2}) = x_{2j+2}$ by (iv). If x_{2j+1} is outer then $b(x_{2j}) = b(x_{2j+1}) \neq x_{2j+1}$. Thus x_{2j+2} is outer and $b(x_{2j+1}) = b(x_{2j+2})$. This completes the induction. ∎

The time for *find_ap_set* is $O(m)$. To show 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 [GT85]. Next note that in a blossom step the vertices u_i are found using this observation: the vertices u'_i are the predecessors of $b(x)$ in the sequence $(b\ell)^j b(y)$, $j = 0, \ldots$ (this follows from properties (*iv*) and (*vii*)). 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 show 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 [G76] 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 time $O(1)$. We now 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 arithmetic operation uses $O(1)$ time.

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

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 $N_s \le 2^{s+1} - 2$. Let Y_s denote the largest magnitude of a y value in scale $s \ge 1$, and set $Y_0 = 0$. It suffices to show the recurrence

$$Y_s \le 2Y_{s-1} + 1 + 5n \log n + 2nN_s.$$

This implies $Y_s \le (2^s - 1)(1 + 5n \log n) + ns2^{s+2}$. Hence in the last scale $Y_s = O(n^2 N \log(nN))$. This implies the desired bound for y.

To prove 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

$$y(\omega) \ge y_0(\omega) - 5n \log n.$$

In *shell_search*(C, D), the value $y(\omega)$ does not change if $\omega \in V(C) - V(D)$, and it decreases by the dual adjustment quantity δ if $\omega \in V(D)$. Thus the total decrease in $y(\omega)$ is at most the total dual adjustment in all *shell_searches* of even shells. As noted after Corollary 6.3, this total is at most $(5/2)\widehat{n}(R)\log\widehat{n}(R)$. Summing over all major path roots R containing ω gives a geometric progression with ratio $1/2$. Thus the total decrease is at most $5n\log n$ as desired.

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

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

Combining this with the above inequality for $y(\omega)$ shows that any vertex x has $y(x) \ge y_0(\omega) - 5n\log n - 2nN_s$. 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 *shell_searches*, but it never increases. In some scale s, let e be a matched edge in the last blossom step (this step forms the blossom G^* and ends the scale). Since G^* is the only blossom containing e, (1*a*) implies that $z(G^*) \ge y(e) - N_s$. Thus $z(G^*)$ satisfies the lemma.

Finally consider any z for nonroot blossoms. These values are nonnegative. Consider a nonroot blossom B. It contains a matched edge e. (1*b*) for e implies that in scale s, $z(B) \le z(\{C \mid e \subseteq C\}) - z(G^*) \le y(e) + N_s + 2 - z(G^*)$. Thus $z(B)$ satisfies the lemma. ▮

All other quantities computed in the algorithm are easily related to y and z. For instance the quantities of Section 7, z_1, t and y', are all easily expressed in terms of $z[G^*, B]$. The last paragraph of the proof shows the latter 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 $y(V(G)) - \lfloor\widehat{n}/2\rfloor z(V(T))$. (This is the objective function of the linear programming dual of the matching problem [E65b].) It is tempting to analyze the matching algorithm using this dual objective function (as done in [G85b]). Here are several easily proved facts: The dual objective does not decrease in *path*. In the entire execution of *path* the dual objective can increase only by

$O(n)$ (by the Double Step). A dual adjustment of δ in the search of a shell containing at least three free vertices increases the dual objective by at least δ. These facts give a good bound on the time spent in *shell_searches* of shells with at least three free vertices. Unfortunately an even shell can 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 already been noted. 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 copies of the same vertex by a cost zero edge e. Minimum cost maximum cardinality matching uses the same construction, except the above edges e have cost nN. ▮

As observed in [G85b], 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 optimum cost perfect matching; here "optimum" is either "maximum" or "minimum". Assume the given cost function c is nonnegative real-valued. We will modify c to a cost function c' which takes on relatively small integral values. We will show that if M (M') is an optimum perfect matching for c (c'), then $c(M')$ is close to $c(M)$.

First consider approximately maximum perfect matching. Fix an integer a. We will define c' to take values in $[0..n^{1+a}]$ and achieve $c(M') \geq (1-n^{-a})c(M)$. Specifically let N be the largest given cost; assume $N > 0$ else the problem is trivial. Define $c' = \lfloor n^{1+a}c/N \rfloor$. Then $c'(M') \geq c'(M)$ implies $n^{1+a}c(M')/N + n/2 > n^{1+a}c(M)/N$. Since $c(M) \geq N$, $c(M') > c(M) - N/n^a \geq (1 - n^{-a})c(M)$, as desired.

Next consider approximately minimum perfect matching. Fix an integer a. We will define c' to take values in $[0..n^{2+a}]$ and achieve $c(M') \leq (1+n^{-a})c(M)$. Let B be the cost of a minimum bottleneck matching, that is, the minimum value such that there is a perfect matching A on the edges costing at most B. Assume $B > 0$ else the problem is trivial. Delete all edges costing more than $c(A)$; clearly such edges are not in a minimum cost matching. Define $c' = \lfloor n^{1+a}c/B \rfloor$. Note that if e is an edge that is not deleted, then $c(e) \leq nB/2$, whence $c'(e) \leq n^{2+a}$ 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 (or at least $(1 - n^{-a})$ times maximum) can be found in $O(a\sqrt{n\alpha(m,n)\log n}\; m \log n)$ time and $O(m)$ space.

Proof. For approximately maximum matching we need only compute c' and use the scaling algorithm for maximum perfect matching. The scaling algorithm runs in the time of the theorem, as desired.

Approximately minimum matching is similar, except we begin by finding a bottleneck matching. A minimum bottleneck matching can be found in $O(\sqrt{n \log nm})$ time [GT88], less than the bound of the theorem. ▮

This leads to an efficient implementation of Christofides' approximation algorithm for a travelling salesman tour. Recall this approximation algorithm works as follows. Given are n cities and the distance between every 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 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 $c(T) + c(M) \leq (3/2)c(H)$. 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 in $O(n^{2.5}(\log n)^{1.5})$ 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 (i.e., edges incident to v are added or deleted, costs of edges incident to v change). Then a new optimum structured matching can be found in the time for one search of Edmonds' algorithm [BD, CM, G85b, W]. Another example is the single source shortest path problem on undirected graphs with no negative cycles [L, 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 $\overline{c}$ is $(2n+2)c$ (in Section 2, $\overline{c} = (n+1)c$). 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

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

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

Observe that for any vertex v,

$$\overline{y}(v) - y_0(v) + Y \in (-n..0].$$

For by definition, $\overline{y}(v) = (2n+2)\lfloor \frac{y_0(v)-Y}{2n+2} \rfloor$. The quantity $Y - y_0(v) - \overline{c}(M_v) \in (-n..0]$, by $(1a)$ – $(1b)$. Since $\overline{c}(M_v)$ is a multiple of $2n+2$ the result follows.

Now we show (i) – (ii). Evaluating the sum $z[G^*, B]$ shows that any edge e_B is tight, i.e., $\widehat{\overline{y}\,\overline{z}}(e_B) = \overline{c}(e_B)$. Since $\widehat{y_0 z_0}(e_B) - \overline{c}(e_B) \in [-2..0]$, subtracting the equality implies $(\overline{z} - z_0)[G^*, B] + 2Y \in [-2n..0]$. Thus for any edge e,

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

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

Lastly consider (iii). Since $(y_0 - \overline{c})e_B - (y_0 - \overline{c})e_A - z_0(B) \in [-2..2]$, the definition of $\overline{z}$ implies that $(\overline{z} - z_0)(B) \in [-2n..2n]$. Thus $z_0(B) \geq 0$ and $\overline{z}(B)$ divisible by $2n + 2$ give the desired conclusion.

Theorem 10.4. An optimum structured matching can be found in the 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. The *scaling routine* is executed with all costs equal to zero, i.e., the *match* routine is called only once. Define ρ, the number of phase 1 iterations of *path*, to be $\lceil\sqrt{n}\rceil$. The remaining details of phase 1 are unchanged. After phase 1 the algorithm is simpler than before. Instead of phases 2–3 it abandons the costs and dual variables and does the following: It repeatedly calls *find_ap_set* to find a maximal set of augmenting paths $\mathcal{P}$; it augments 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. In phase 1 of the cardinality matching algorithm, at any time in *path*(G^*), $\widehat{\delta}(F_\omega, G^*) \leq n$.

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 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 $z(G^*) = -2\widehat{\delta}(G^*)$, $\mu(G^*) = |F_\omega|/2$, and $|M| \leq n/2$ imply the lemma. ▮

At the end of phase 1, any free vertex v has $\widehat{\delta}(v, G^*) = \rho$. Thus $|F_\omega|\rho \leq n$. This implies 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 before 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 that of Micali and Vazirani [MV]. Note that our algorithm is not the same as theirs: For instance it operates with inner blossoms effectively "shrunk" in phase 1. Also our depth-first search may involve less overhead than the "double depth-first search" of [MV].

Finally note that in practice a different organization after phase 1 is probably faster: The algorithm calls $find\_ap(x)$ for each free vertex x. If the latter does not find an augmenting path then the vertices it scanned are still marked "scanned" in subsequent searches. This works correctly because these vertices cannot be in augmenting paths (see [G76]).

11. Concluding remarks.

The matching algorithm generalizes to degree-constrained subgraphs. Consider a graph having two functions $\ell, u : V \to \mathbf{Z}$. A *degree-constrained subgraph* (*DCS*) is a subgraph where each vertex v has degree in the range $[\ell(v)..u(v)]$. 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 [G87]. 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 [L]. 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 (the underlying complete graph of) a set of $2n$ points can be found in time $n^{2.5}\log^{O(1)}n$ and space $O(n\log n)$ [V]. Furthermore it appears that applying our algorithm reduces the time by a factor of about $\sqrt{n}$ [V].

Figure Captions

References.

[AHU] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[Ber86] D.P. Bertsekas, "Distributed asynchronous relaxation methods for linear network flow problems", LIDS Report P-1606, M.I.T., Cambridge, Mass.,1986; preliminary version in *Proc.* 25th *IEEE Conf. on Decision and Control*, December 1986.

[BD] M.O. Ball and U. Derigs, "An analysis of alternative strategies for implementing matching algorithms", *Networks 13*, 4, 1983, pp. 517-549.

[C] N. Christofides, "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.

[CM] W.H. Cunningham and A.B. Marsh, III, "A primal algorithm for optimum matching", *Math. Programming Study 8*, 1978, pp. 50-72.

[E65a] J. Edmonds, "Paths, trees and flowers", *Canad. J. Math. 17*, 1965, pp. 449-467.

[E65b] J. Edmonds, "Maximum matching and a polyhedron with 0,1-vertices", *J. Res. Nat. Bur. Standards 69B*, 1965, 125-130.

[G76] H.N. Gabow, "An efficient implementation of Edmonds' algorithm for maximum matching on graphs", *J. ACM 23,* 2, 1976, pp. 221-234.

[G85a] H.N. Gabow, "Scaling algorithms for network problems", *J. of Comp. and Sys. Sciences 31*, 2, 1985, pp. 148-168.

[G85b] H.N. Gabow, "A scaling algorithm for weighted matching on general graphs", *Proc.* 26th *Annual Symp. on Foundations of Comp. Sci.,* 1985, pp. 90-100.

[G87] H.N. Gabow, "Duality and parallel algorithms for graph matching", manuscript.

[Go] A.V. Goldberg, "Efficient graph algorithms for sequential and parallel computers", Ph. D. Dissertation, Dept. of Electrical Eng. and Comp. Sci., MIT, Technical Rept. MIT/LCS/TR-374, Cambridge, Mass., 1987.

[GoT87a] A.V. Goldberg and R.E. Tarjan, "Solving minimum-cost flow problems by successive approximation", *Proc.* 19th *Annual ACM Symp. on Th. of Computing*, 1987, pp. 7-18.

[GoT87b] A.V. Goldberg and R.E. Tarjan, "Finding minimum-cost circulations by successive approximation", Technical Rept. CS-TR-106-87, Department of Comp. Sci., Princeton

University, Princeton, New Jersey, 1987.

[GGS] H.N. Gabow, Z. Galil, T.H. Spencer, "Efficient implementation of graph algorithms using contraction", *Proc. 25th Annual Symp. on Found. of Comp. Sci.,* 1984, pp.347-357.

[GMG] Z. Galil, S. Micali, and H.N. Gabow, "An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs", *SIAM J. Comp. 15,* 1, 1986, pp. 120-130.

[GT85] H.N. Gabow and R.E. Tarjan, "A linear-time algorithm for a special case of disjoint set union", *J. of Comp. and Sys. Sciences 30*, 2, 1985, pp. 209–221.

[GT87] H.N. Gabow and R.E. Tarjan, "Faster scaling algorithms for network problems", *SIAM J. Comp.*, to appear.

[GT88] H.N. Gabow and R.E. Tarjan, "Algorithms for two bottleneck optimization problems", *J. Algorithms 9*, 3, 1988, pp. 411–417.

[H] F. Harary, *Graph Theory,* Addison-Wesley, Reading, Mass., 1969.

[HK] J. Hopcroft and R. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs", *SIAM J. Comp. 2*, 4, 1973, pp. 225-231.

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

[MV] S. Micali and V.V. Vazirani, "An $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding maximum matching in general graphs", *Proc. 21st Annual Symp. on Found. of Comp. Sci.*, 1980, pp. 17-27.

[PS] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[T79] R.E. Tarjan, "Applications of path compression on balanced trees", *J. ACM 26*, 4, 1979, pp. 690-715.

[T83] R.E.Tarjan, *Data Structures and Network Algorithms*, SIAM Monograph, Philadelphia, Pa., 1983.

[Tard] É. Tardos, "A strongly polynomial minimum cost circulation algorithm", *Combinatorica 5,* 3, 1985, pp. 247–255.

[V] P.M. Vaidya, "Geometry helps in matching", *Proc. Twentieth Annual ACM Symp. on Th. of Computing*, 1988, pp. 422-425.

[W] G.M. Weber, "Sensitivity analysis of optimal matchings", *Networks 11*, 1981, pp. 41-56.

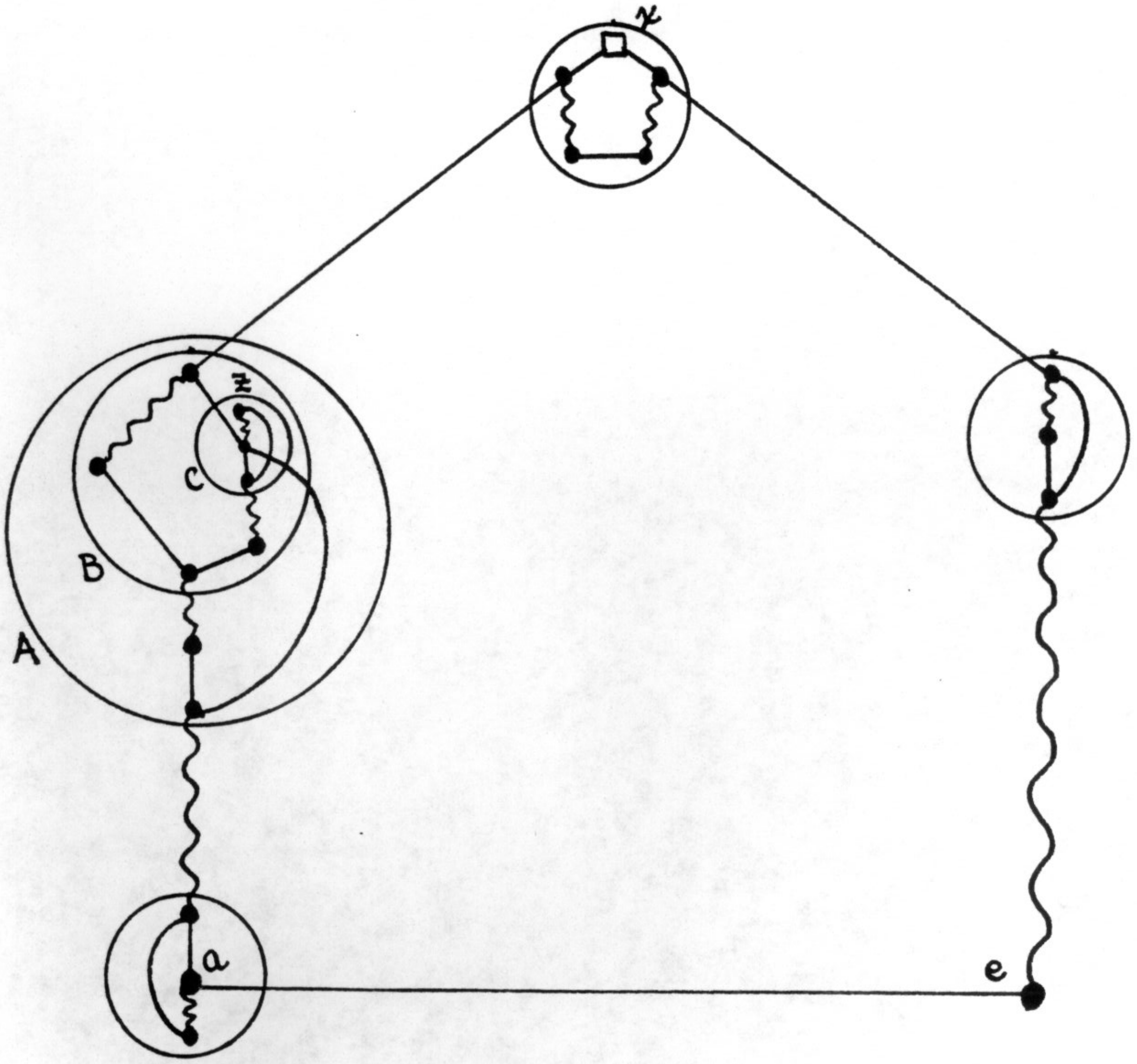

Fig. 1.1

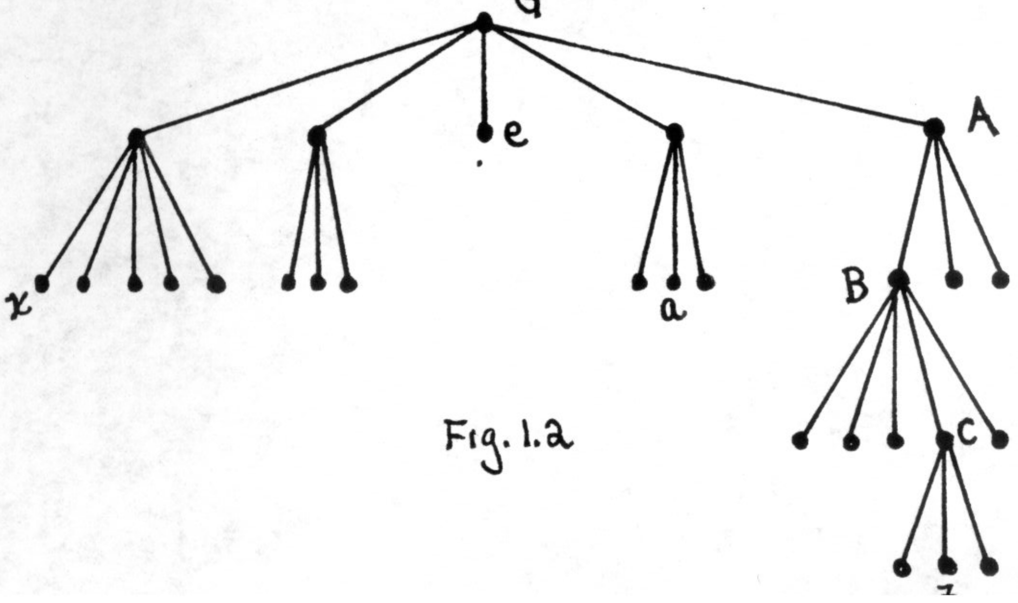

Fig. 1.2

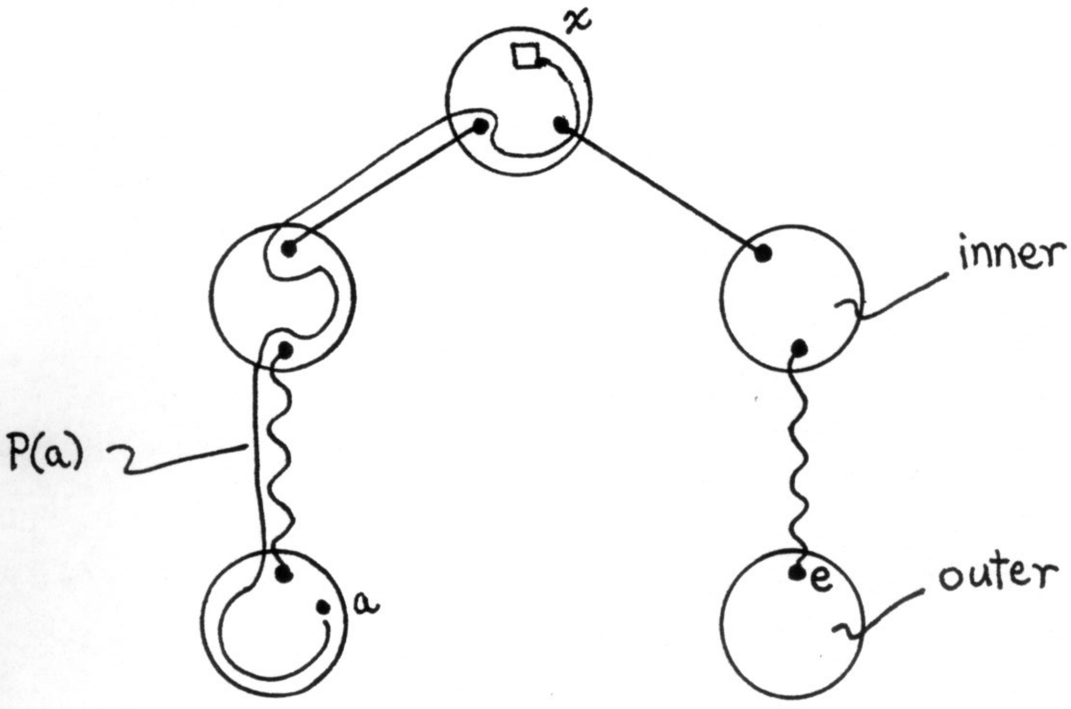

Fig. 1.3

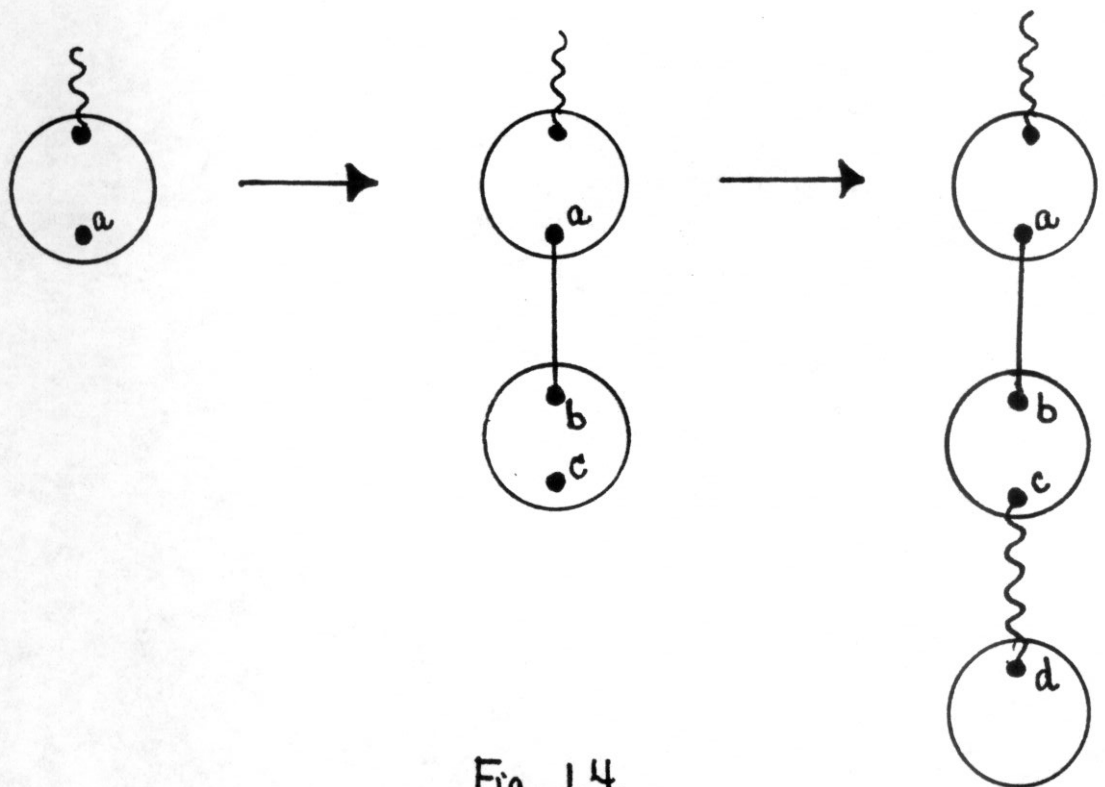

Fig. 1.4.

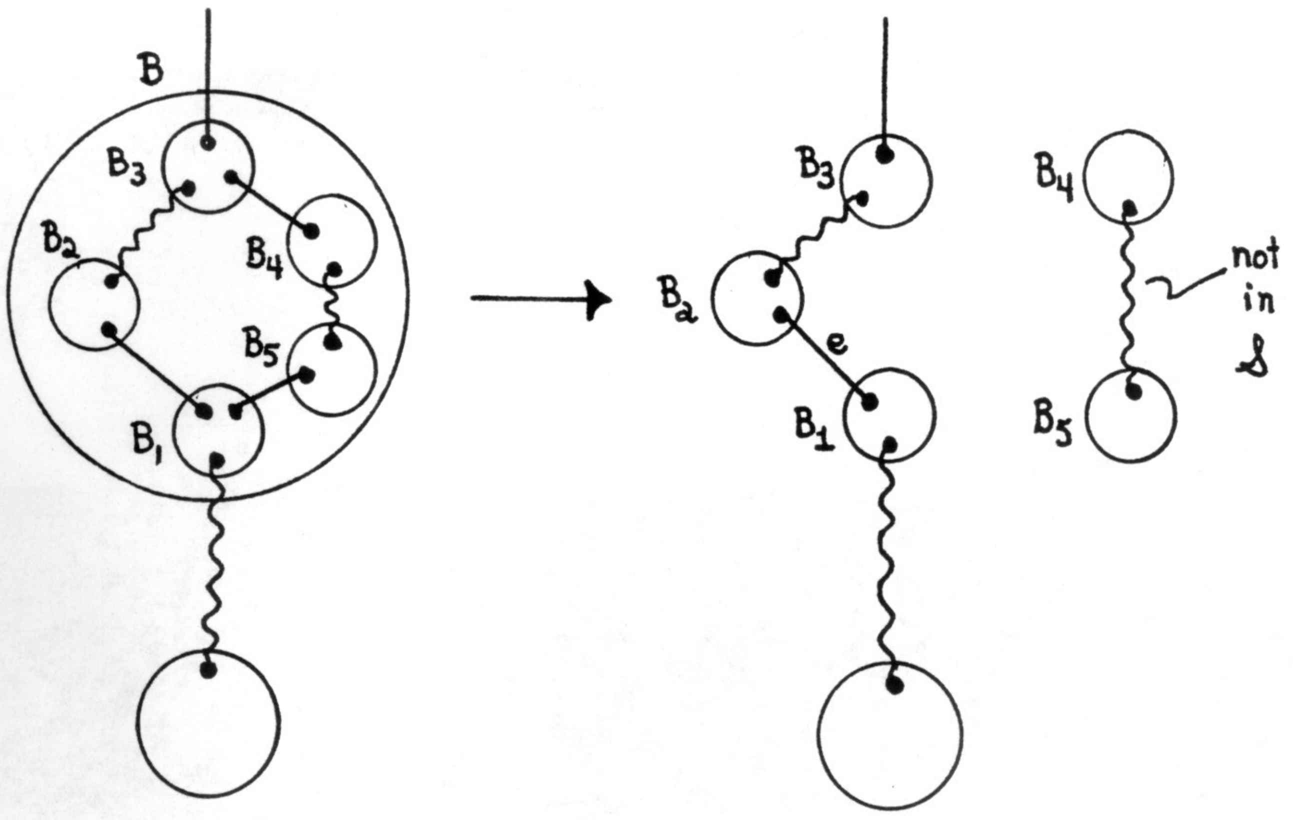
B
B_3
B_2
B_4
B_5
B_1
B_3
B_4
B_2
e
not in $\mathcal{S}$
B_1
B_5

Fig. 1.5

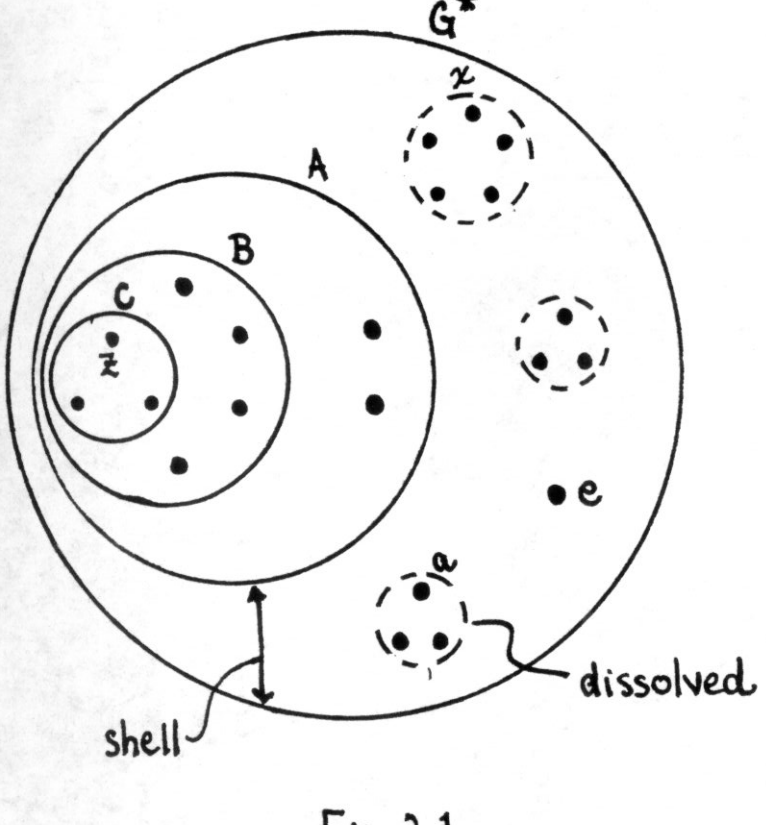
G^*
x
A
B
C
z
e
a
dissolved
shell

Fig. 3.1

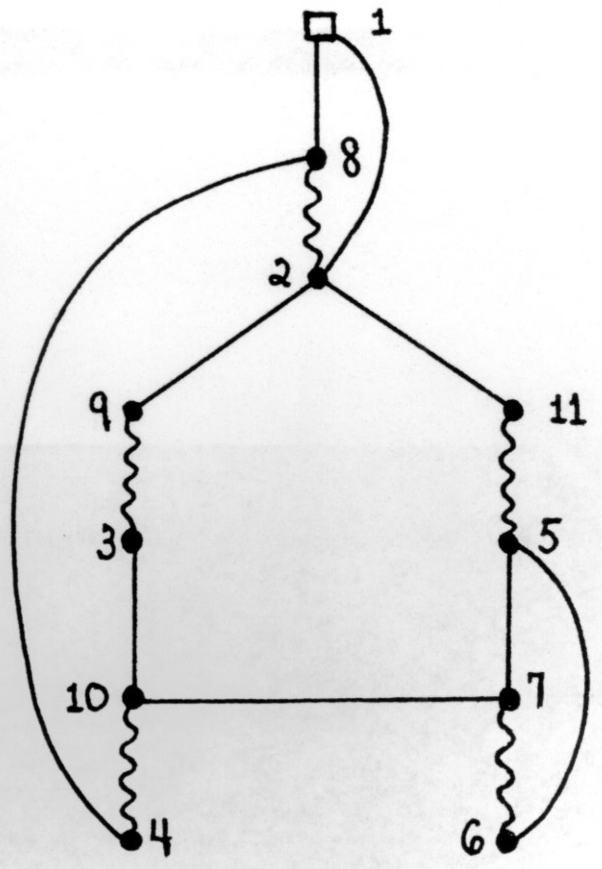

Fig. 8.1

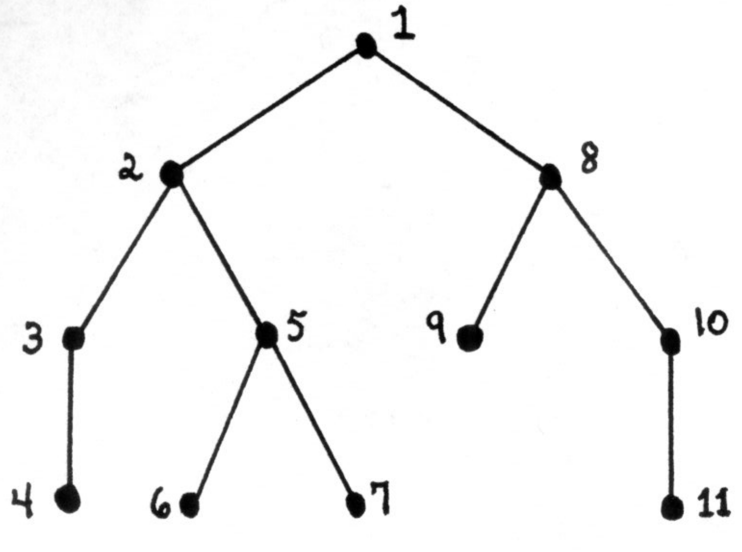

Fig. 8.2