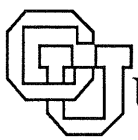
Data Structures for Weighted Matching and Nearest Common Ancestors with Linking

Harold N. Gabow

CU-CS-478-90 June 1990



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Data Structures for Weighted Matching and Nearest Common Ancestors with Linking

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

June 26, 1990

Abstract. This paper shows that the weighted matching problem on general graphs can be solved in time $O(n(m + n\log n))$, for n and m the number of vertices and edges, respectively. This was previously known only for bipartite graphs. The paper also shows that a sequence of m *nca* and *link* operations on n nodes can be processed on-line in time $O(m\alpha(m,n)+n)$. This was previously known only for a restricted type of *link* operation.

---

[1] Research supported in part by NSF Grant No. CCR-8815636.

1. Introduction.

This paper solves two well-known problems in data structures and gives some related results. The starting point is the matching problem for graphs, which leads to the other problems. This section defines the problems and states the results.

A *matching* on a graph is a set of vertex-disjoint edges. Suppose each edge e has a real-valued *cost* $c(e)$. The cost $c(S)$ of a set of edges S is the sum of the individual edge costs. A *minimum-cost matching* is a matching of smallest possible cost. There are a number of variations: a *minimum-cost maximum cardinality matching* is a matching with the greatest number of edges possible, which subject to this constraint has the smallest possible cost; *minimum-cost cardinality-k matching* (for a given integer k), *maximum-weight matching*, etc., are defined similarly. The *weighted matching problem* refers to all of the problems in this list.

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

Weighted matching is a classic problem in network optimization; detailed discussions are in [L, LP, PS]. Edmonds gave the first polynomial-time algorithm for weighted matching [E]. Several implementations of Edmonds' algorithm have been proposed, with increasingly fast running times: $O(n^3)$ [G73, L], $O(mn \log n)$ [BD, GMG], $O(n(m \log\log\log_{2+m/n} n + n \log n))$ [GGS]. Edmonds' algorithm is a generalization of the Hungarian algorithm, due to Kuhn, for weighted matching on bipartite graphs [K55, K56]. Fredman and Tarjan implement the Hungarian algorithm in $O(n(m + n \log n))$ time using Fibonacci heaps [FT]. They ask if general matching can be done in this time. Our first result is an affirmative answer: We show that a search in Edmonds' algorithm can be implemented in time $O(m+n \log n)$. This implies that the weighted matching problem can be solved in time $O(n(m + n \log n))$. In both cases the space is $O(m)$. Our implementation of a search is in some sense optimal: As shown in [FT] for Dijkstra's algorithm, one search of Edmonds' algorithm can be used to sort n numbers. Thus a search requires time $\Omega(m + n \log n)$ in an appropriate model of computation.

Another algorithm for weighted matching is based on cost scaling. This approach is applicable if all costs are integers. The best known time bound for such a scaling algorithm is $O(\sqrt{n\alpha(m,n)\log n}\ m \log(nN))$ [GT89]; here N is the largest magnitude of an edge cost and α is an inverse of Ackermann's function (see Section 3). Under the similarity assumption [G85a] $N \leq n^{O(1)}$, this bound is superior to Edmonds' algorithm. However our result is still of interest for at least two reasons: First, Edmonds' algorithm is theoretically attractive because its time bound

is strongly polynomial. Second, for a number of matching and related problems, the best known solution amounts to performing one search of Edmonds' algorithm. These problems include most forms of sensitivity analysis for weighted matching [BD, CM, G85b, W] and the all-pairs shortest path problem on undirected graphs with no negative cycles [G83, L]. Thus our implementation of a search in time $O(m + n \log n)$ gives the best-known algorithm for these problems.

An important step in solving the matching problem is to solve a nearest common ancestor problem on trees. This is our second contribution. To define such problems recall that the *nearest common ancestor* of two nodes x and y in a tree is the ancestor of x and y that has greatest depth. Consider a forest F that is subject to two operations. In both operations x and y are nodes of F.

link(x, y) – make y a child of x, where y is the root of a tree not containing x;

nca(x, y) – return the nearest common ancestor of x and y (if x and y are in different trees, return $\emptyset$).

The problem of *nearest common ancestors with linking* is to process (on-line) an arbitrary sequence of m *link* and *nca* operations, starting from an initial forest of n nodes ($m, n \geq 1$).

A number of data structures have been proposed for this problem and various special cases; see [HT] for a more complete summary. The problem of *nearest common ancestors for static trees* is when F is given initially (equivalently, all *links* precede all *ncas*). Harel and Tarjan give an algorithm that answers each *nca* query in time $O(1)$, after $O(n)$ time to preprocess F [HT]. (More recently [SV] uses another approach to achieve the same result sequentially by an algorithm that can be parallelized.) [HT] also gives an algorithm for the case where *link* and *nca* operations are intermixed, but both arguments of every *link* are roots. The running time is $O(m\alpha(m, n) + n)$. Our second contribution is to remove the restriction: We show that the general nearest common ancestors with linking problem can be solved in time $O(m\alpha(m, n) + n)$ and space $O(n)$. The previous best solution is using dynamic trees [SlT]. This data structure performs each operation in time $O(\log n)$, achieving total time $O(m \log n + n)$. This is not as fast as our algorithm, but dynamic trees have the advantage that they can also process *cut* operations. Again our result is in some sense optimal: Mike Fredman has pointed out that the results of [FS] can be extended to show that nearest common ancestors with linking requires time $\Omega(m\alpha(m, n) + n)$ in the cell probe model of computation.

Edmonds' matching algorithm does not require fully general *link* operations. Consider the following two special cases of *link*. They operate on a single tree T; x denotes a node already in T, y is a new node not yet in T:

add_leaf(x, y) – add a new leaf y, with parent x, to T;

add_root(y) – make the current root of T a child of new root y.

The problem of *incremental-tree nearest common ancestors* is to process (on-line) an arbitrary sequence of n *add_leaf* and *add_root* operations intermixed with m *nca* operations, starting with an empty tree. We show this can be done in time $O(m+n)$. This is the starting point of our algorithm for general *links*. Edmonds' algorithm actually only uses *add_leaf* and *nca* operations.

The model of computation throughout this paper is a random access machine with a word size of $\log n$ bits. [HT] gives a lower bound indicating it is unlikely that our results for nearest common ancestors can be achieved on a pointer machine. On the other hand it still might be possible to achieve our results for Edmonds' algorithm on a pointer machine.

Section 2 gives our implementation of Edmonds' algorithm. Section 3 gives our algorithm for the nearest common ancestor problem. This section concludes with some terminology and assumptions.

We use interval notation for sets of integers: for integers i and j, $[i..j] = \{k|\ k$ is an integer, $i \leq k \leq j\}$, and similarly for $[i..j)$, etc. The function *log* n denotes logarithm to the base two. The function $log^{(i)}n$ denotes the iterated logarithm, defined by $\log^{(1)}n = \log n$, $\log^{(i+1)}n = \log(\log^{(i)}n)$. In contrast $log^i n$ denotes $\log n$ raised to the ith power.

We assume that for a given integer $s \in [1..n]$ the value $\lfloor \log s \rfloor$ can be computed in $O(1)$ time. This can be done if we precompute these n values and store them a table. The precomputation time is $O(n)$. This can be ignored since it is dominated by the time bounds for our algorithms.

For a graph G, $V(G)$ denotes its vertices and $E(G)$ its edges. We use the following terminology for trees throughout the paper. Let T be a tree. A subtree of T is a connected subgraph. The root of T is denoted $r(T)$. Let v be a node of T. The *ancestors* of v are the nodes on the path from v to $r(T)$. The ancestors are ordered as in this path. This indicates how to interpret expressions like "the first ancestor of v such that". The *parent* of v is denoted $p(v)$. The *depth* of v, denoted $d(v)$, is the length of the path from v to $r(T)$ (equivalently, the number of proper ancestors of v). For any function f defined on nodes of a tree, we write f_T when the tree T is not obvious (e.g., $p_T(v)$, $nca_T(x, y)$).

2. Weighted matching.

This section presents an implementation of Edmonds' weighted matching algorithm with run time $O(n(m + n\log n))$. This is accomplished in three steps. Section 2.1 summarizes Edmonds' algorithm and our reduction to the tree-blossom merging problem. Section 2.2 gives an algorithm

for tree-blossom merging that comes close to the desired efficiency. Section 2.3 refines the merging algorithm to achieve the desired time and space bounds. (Both merging algorithms use additional tree operations discussed in Section 3.) This section concludes with some additional terminology for matching. Fix a matching M on the given graph. A vertex is *free* if it is not on any matched edge. An *alternating path* is a vertex simple path whose edges are alternately matched and unmatched. An *augmenting path* P is an alternating path joining two distinct free vertices. To *augment the matching along* P means to enlarge the matching M to $M \oplus P$ (the symmetric difference of M and P). This gives a matching with one more edge.

2.1. Edmonds' algorithm and tree-blossom merging.

This section first sketches Edmonds' algorithm. It gives the details needed for our implementation but omits other material. Complete descriptions of the algorithm are in [E, G73, L, PS]. The section then defines the tree-blossom merging problem and shows how it leads to an efficient implementation of Edmonds' algorithm.

The matching algorithm is a primal-dual algorithm based on Edmonds' formulation of weighted matching as a linear program. It works by repeatedly finding a maximum weight augmenting path and using it to enlarge the matching. The procedure to find an augmenting path is called a *search*. If the search is successful, i.e., it finds an augmenting path P, then an *augment step* is done. It augments the matching along P. The exact definition of unsuccessful search depends on the variant of the matching problem being solved and does not concern us here; see [L]. The entire algorithm consists of $O(n)$ searches and augment steps.

A search constructs a subgraph $\mathcal{S}$. $\mathcal{S}$ is initialized in a simple way that depends on the variant of matching. (For instance one initialization sets the root of $\mathcal{S}$ to a blossom containing a free vertex; see below for the meaning of these terms.) Then $\mathcal{S}$ is enlarged by executing three types of steps, called *grow*, *blossom* and *expand steps* in [G85b]. In addition the search changes the linear programming dual variables in *dual adjustment steps*. After a dual adjustment step, one or more of the other steps can be performed. Steps are repeated until $\mathcal{S}$ contains the desired augmenting path, or the search is deemed unsuccessful.

A *blossom* is a special type of subgraph. We need not give a precise definition of blossom for this paper. It suffices to note that at any time in Edmonds' algorithm the vertices of the graph are partitioned into blossoms. (A single vertex can be a blossom. At the start of the algorithm the partition is the trivial one consisting of all singleton blossoms. However in general at the start of a search the partition is nontrivial.) A blossom step forms a new blossom, as illustrated in Figure

1. Here edges are drawn wavy if they are matched and straight otherwise. The circles B_i represent blossoms at some time in the search; at that time $\mathcal{S}$ contains the entire subgraph shown except edges e, f and g ($\mathcal{S}$ also contains other edges not shown). When e gets added to $\mathcal{S}$ a blossom step is done for e. It forms a new blossom B containing the entire subgraph shown. (The blossoms B_i cease to exist.)

In general $\mathcal{S}$ consists of blossoms arranged in a tree structure. More precisely if each blossom is contracted to a vertex, $\mathcal{S}$ becomes a tree $\overline{\mathcal{S}}$. (This assumes the simple initialization mentioned above. Other initializations make $\overline{\mathcal{S}}$ a forest. We assume the tree case for simplicity, but our algorithm extends trivially to forests.) A blossom of $\mathcal{S}$ that is an even (odd) distance from the root of $\overline{\mathcal{S}}$ is *outer* (*inner*); a vertex of $\mathcal{S}$ is outer or inner, depending on the blossom that contains it. In Figure 1 the outer blossoms are the B_i with i even before the blossom step for e, and B after. Vertices u, v and w are inner before the blossom step and outer after. Now we discuss the three steps that construct $\mathcal{S}$ in turn.

A grow step enlarges $\mathcal{S}$ by adding an inner blossom and an outer blossom. In Figure 1 if $\mathcal{S}$ contains B_0 but no other B_i, a grow step for edge a adds a, B_1, b and B_2 to $\mathcal{S}$; similar grow steps add the other B_i. (In general a special case of the grow step is when the new inner blossom is not matched to another, i.e., it contains a free vertex. In this case an augmenting path has been found. The search is successful so it halts.)

A blossom step is executed for an edge e joining two distinct outer blossoms. It combines all blossoms along the fundamental cycle of e in the tree $\overline{\mathcal{S}}$ to form a new outer blossom. This destroys the old blossoms along the cycle. In Figure 1, when $\mathcal{S}$ contains all blossoms B_i, $i = 0, \ldots, 6$, the search might do a blossom step for e, as already described. Alternatively it might do a blossom step for g, then later one for f, and still later one for e. Other sequences are possible. The sequence executed depends on the costs of these edges.

An expand step replaces an inner blossom in $\mathcal{S}$ by some of its subblossoms. In Figure 1 when $\mathcal{S}$ contains all blossoms B_i, an expand step for blossom B_1 replaces it by vertices u, v, w and edges uv, vw. This preserves the tree structure of $\overline{\mathcal{S}}$, keeping previous descendants of B_1 in the tree. The two other vertices of B_1 leave $\mathcal{S}$. Note that because of expand steps a vertex can alternate between being in $\mathcal{S}$ and inner, and being not in $\mathcal{S}$. This alternation can occur arbitrarily many times in one search. However once a vertex becomes outer it remains in $\mathcal{S}$ and outer for the rest of the search.

The precise sequence of these steps executed by the search depends on the costs of edges and the values of the dual variables. The algorithm executes a certain step when a corresponding edge becomes "tight", i.e., its dual variables satisfy the complementary slackness conditions of

linear programming. For example in Figure 1 the blossom step for e is done when e becomes tight. The dual adjustment step modifies dual variables so that additional edges become tight and corresponding grow, blossom or expand steps can be done. The dual adjustment step involves finding a candidate edge that is closest to being tight, and then adjusting the duals to make it tight.

Our task is to implement a search in time $O(m + n \log n)$. (In the case of a successful search, we implement the corresponding augment step in time $O(n)$.) It is known how to implement most details of a search within the desired time bound. For dual adjustment steps a Fibonacci heap $\mathcal{F}$ is used. It contains the candidate edges mentioned above. The heap minimum gives the next edge to be made tight. This is analogous to the implementation of Dijkstra's algorithm by Fibonacci heaps, and uses the same time per search, $O(m + n \log n)$ [FT]. In addition dual variables are maintained in time $O(n)$ per search using offset quantities [e.g., GMG]. The processing associated with grow and expand steps can be done in time $O(m\alpha(m, n))$ using a data structure for list splitting given in [G85b]. This bound is dominated by $O(m + n \log n)$ [T83]. Finally note after a successful search, the corresponding augment step can be done in time $O(n)$ [GMG]. This leaves only the blossom steps: implementing the blossom steps of a search in time $O(m + n \log n)$ gives the desired result. (This observation is also made in [GGS].)

The problem of implementing the blossom steps of a search can be stated precisely as the *blossom merging problem*, which we now define. ([GGS] defines a similar problem, called on-line restricted component merging.) The universe consists of a graph with vertex set O and edges $\mathcal{E}$. Set O corresponds to the set of outer vertices in Edmonds' algorithm. It is initially empty. At any time it is partitioned into subsets called blossoms. The edge set $\mathcal{E}$ is initially empty. At any time it contains all edges that join two outer vertices in Edmonds' algorithm. Thus $\mathcal{E}$ includes all edges that can cause blossom steps. The problem is to process (on-line) a sequence of the following types of operations:

make_blossom(A) – add the set of vertices A to O and make A a blossom (this assumes that $A \cap O = \emptyset$ before the operation);

make_edge(vw, t) – add edge vw, with cost t, to $\mathcal{E}$ (this assumes $v, w \in O$);

merge(A, B) – combine blossoms A and B into a new blossom (this destroys the old blossoms A and B);

find_min – return an edge vw of $\mathcal{E}$ that has minimum cost subject to the constraint that v and w are (currently) in distinct blossoms.

Let us sketch how these four operations are used to process blossom steps in Edmonds' algorithm. Grow, expand and blossom steps each create new outer blossoms. They perform *make_blossom* operations to add the new outer vertices to O. They also perform *make_edge* operations for the new edges that join two outer vertices. For example in Figure 1 if B_0, B_5 and B_6 are in $\mathcal{S}$ and a grow step is done for edge a then *make_blossom*(B_2) is done; also *make_edge* is done for edge f. Note that in *make_edge*(vw, t), t is not the given cost $c(vw)$. Rather t is $c(vw)$ modified by dual values; this modification allows the algorithm to make dual adjustments efficiently [GMG]. The value of t is unknown until the time of the *make_edge* operation. From now on, since we are only concerned with the blossom merging problem, the "cost" of an edge of $\mathcal{E}$ refers to this value t, not the cost input to the matching algorithm.

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

A *find_min* operation is done at the end of each of the three search steps. The returned edge is used in the above-mentioned Fibonacci heap $\mathcal{F}$ to select the next step of the search. In Figure 1 if e is returned by *find_min*, it gets added to $\mathcal{F}$. Note that for correctness of the implementation it is important that the blossom merging problem uses a *find_min* operation rather than *delete_min*: The current minimum edge e for a future blossom step may not be selected (in the dual adjustment step) as the next tight edge for Edmonds' search algorithm. Instead if a different edge is selected, the next step of the search may create new blossom steps for edges smaller than e. In this case it would be premature to delete e from the data structure when it is found to be minimum.

Our task is to implement a sequence of these operations – *make_blossoms* adding a total of n vertices, m *make_edges*, at most n *merges* and at most n *find_mins*, in time $O(m + n \log n)$. (The bound on *find_mins* follows since *find_min* need not be done after an expand step that does not create a new outer vertex. Every other step creates a new outer vertex (a blossom step changes an inner vertex to outer). So there are at most n such steps and n corresponding *find_mins*.) The difficulty in solving this blossom merging problem is illustrated by Figure 1. When each B_i, $i = 0, \ldots, 6$, is a blossom, edges e, f, g are candidates for *find_min*. If a blossom step for e is done,

edges f and g become irrelevant, since they no longer join distinct blossoms. If we were to store the edges of $\mathcal{E}$ in a priority queue, we would end up with useless edges like f, g in the queue. These edges would eventually get deleted from the queue but the deletions accomplish no useful work. This could make the algorithm spend too much time, $\Theta(m \log n)$. (This also indicates why there is no need for a *delete_min* operation in the blossom merging problem: If edge e of Figure 1 gets returned by *find_min* then after the blossom step forming B, edge e is irrelevant.)

Algorithms for general blossom merging are given in [GGS]. Our solution is to define a special case of the blossom merging problem that incorporates additional information from Edmonds' algorithm. To do this we wish to maintain a representation of the search graph $\mathcal{S}$ by a tree T. For reasons of efficiency T should be constructed by simple operations (we will use the operation of adding a leaf). This can be done if some care is taken for expand steps. Recall the expand step illustrated in Figure 1, where blossom B_1 gets replaced by the path u, v, w. If our tree T were to represent B_1 as a vertex then the expand step would have to replace that vertex by the new path u, v, w. The tree algorithms of Section 3 do not process such "splice" operations. The solution is to define T so that at all times B_1 is represented by path u, v, w, so that splices are not needed.

To do this in general we first recall another detail of Edmonds' algorithm. Consider a blossom B in the search graph $\mathcal{S}$. Suppose $\overline{\mathcal{S}}$ contains a matched edge incident to B; let that edge be incident to vertex $w \in B$ (in Figure 1 see vertex w in B_1). Any vertex $x \in B$ is joined to w by an even-length alternating path of edges in B, which we denote as $P(x, w)$. (In Figure 1, $P(u, w)$ is the length two path u, v, w; $P(v, w)$ is the counterclockwise path of length four.) Edmonds' algorithm maintains a data structure to calculate these paths. (These paths are needed because they make up the augmenting path found by the search. For example in Figure 1 if an edge from B_2 to a free vertex is added to the search graph it completes an augmenting path, that includes edges a and b and the path joining them $P(u, w)$.) A tree-based data structure that can be used to compute a path $P(x, w)$ in time proportional to its length is given in [GMG, GT89].

Now suppose blossom B is inner and it gets expanded. As in Figure 1, before the expansion $\overline{\mathcal{S}}$ contains edges a and b, incident to vertices u and w of B respectively. The expand step replaces B by the subblossoms along the path $P(u, w)$. In Figure 1 these subblossoms are simply vertices, but in general they can be large blossoms. So in general an expand step produces new outer blossoms and new inner blossoms. The new inner blossoms can be subsequently expanded.

No matter how many blossoms are expanded, $P(u, w)$ always remains in $\mathcal{S}$. This can be shown by a simple induction: Suppose similar to Figure 1, path $P(u, w)$ enters a subblossom B' along an unmatched edge a', with end $u' \in B'$, and it leaves B' along a matched edge b', with end $w' \in B'$.

By definition $P(u,w)$ traverses B' along $P(u',w')$. (We do not supply a precise definition of $P(u,w)$ in this paper, but it can be found in [GMG, GT89] and others.) B' is a new inner blossom. When it gets expanded it is replaced by the subblossoms along the path $P(u',w')$. This completes the induction.

Now we show how to construct the tree T. We use operations *add_leaf*(x,y) (defined in Section 1). First consider grow steps. Suppose a grow step enlarges $\mathcal{S}$ by adding, as illustrated in Figure 1, an unmatched edge a, an inner blossom B_1, a matched edge b and an outer blossom B_2; let a and b be incident to vertices u and w in B_1, respectively, and also to vertices u' and w' not in B_1, respectively. First calculate $P(u,w)$. Denote this path as u_i, $i = 1,\ldots,k$, where $u_1 = u$, $u_k = w$. Then enlarge T by performing the operations *add_leaf*(u',u_1), *add_leaf*(u_i,u_{i+1}) for $i = 1,\ldots,k-1$, *add_leaf*(u_k,w') and finally *add_leaf*(w',x) for all vertices $x \in B_2 - w'$.

Next consider expand steps. Using the above notation, the expand step replaces inner blossom B by the subblossoms along the path $P(u,w)$. For each new outer blossom B', let x be a vertex in B' that is in $P(u,w)$ and hence already in T; do *add_leaf*(x,y) for every vertex y in B' but not yet in T.

Finally consider a blossom step, say for edge vw. It combines the blossoms on the fundamental cycle C of vw in $\overline{\mathcal{S}}$. Perform *add_leaf*(v,u) for every vertex u that is in an inner blossom of C but not yet in T.

This completes the algorithm for constructing T. It is clear that T contains all the outer vertices of $\mathcal{S}$ plus a subset of the inner vertices. (T may not contain some inner vertices, e.g., in Figure 1 it contains only three of the five vertices of B_1.) Now we prove the main property of T.

Lemma 2.1. The vertices of any blossom of $\mathcal{S}$ form a subtree (i.e., a connected subgraph) of T.

Proof. The lemma is clear for an inner blossom B when it gets added to $\mathcal{S}$ and T. Observe that, in greater detail, the vertices in $B \cap T$ form a path in T, and only the deepest vertex has a child not in B. It is clear that the lemma and in fact this observation continue to hold after expand steps.

To prove the lemma for outer blossoms we first introduce some notation. In this proof only, if x is a vertex in a tree X let $p(x,X)$ denote the path from x to the root of X. Thus a blossom B in $\mathcal{S}$ has a path $p(B,\overline{\mathcal{S}})$, whose vertices are blossoms. A vertex $v \in T$ has the path $p(v,T)$. Let $\overline{p}(v,T)$ denote the sequence of blossoms along $p(v,T)$. (More precisely $\overline{p}(v,T)$ is the sequence B_i, $i = 1,\ldots,k$, where $p(v,T)$ starts with a positive number of vertices in B_1, followed by a positive number of vertices of B_2, etc.) We prove the lemma for outer vertices, along with the following property:

(i) For an outer blossom B and a vertex $v \in B$, $p(B,\overline{\mathcal{S}}) = \overline{p}(v,T)$.

The proof is by induction on the number of steps. First consider a grow step. Use the above notation describing how T is constructed in a grow step. Clearly the vertices in the new outer blossom B_2 form a subtree of T. To verify (i), consider a new outer vertex $x \in B_2$. Then $p(x,T)$ goes from x, a vertex of B_2, to u_i, $i > 1$, vertices of B_1, and then follows $p(u_0,T)$. Let u_0 be in the outer blossom B_0. The $p(B_2,\overline{\mathcal{S}})$ is the sequence B_2, B_1, $p(B_0,\overline{\mathcal{S}})$. The inductive assertion for u_0 gives the desired equality. Expand steps are analyzed similar to grow steps.

Consider a blossom step, say for edge vw. Let v and w be in blossoms V_1 and W_1 respectively. Let $p(V,\overline{\mathcal{S}}) = V_i$, $i = 1,\ldots,k$, and $p(W,\overline{\mathcal{S}}) = W_j$, $j = 1,\ldots,\ell$. Let the nearest common ancestor of V_1 and W_1 in $\overline{\mathcal{S}}$ be $V_r = W_s$. Thus the blossom step for vw combines all blossoms V_i, $i = 1,\ldots,r$ and W_j, $j = 1,\ldots,s$ into a new blossom X.

Let a be the nearest common ancestor of v and w in T. By induction, property (i) shows that $a \in V_r \cap W_s$; furthermore the blossoms in X are precisely those intersected by the paths from v to a and w to a in T. This gives the lemma for the new blossom X.

To show (i) observe that $p(X,\overline{\mathcal{S}})$ is the sequence X followed by V_i, $i = r+1,\ldots,k$. Consider first a vertex $x \in X$; let x be in blossom V_p before the blossom step. (The argument for $x \in W_p$ is identical.) Suppose V_p is outer before the blossom step. The inductive assertion shows that $p(x,T)$ goes through blossoms V_q, $q = p,\ldots,k$. Thus after the blossom step $p(x,T)$ goes through X, V_i, $i = r+1,\ldots,k$ and (i) holds for x. In particular (i) holds for v. We now show that if V_p is inner before the blossom step then $\overline{p}(x,T) = \overline{p}(v,T)$, giving ($i$) for x. If x is not in T before the blossom step then obviously $\overline{p}(x,T) = \overline{p}(v,T)$. If x is in T before the blossom step then $p(v,T)$ contains x. (This follows from the observation made in the discussion of the lemma for inner blossoms.) So again $\overline{p}(x,T) = \overline{p}(v,T)$.

Finally consider a blossom B that is disjoint from X. The path $p(B,\overline{\mathcal{S}})$ traverses the same blossoms before and after the blossom step, except that some subpath, of nonnegative length, changes from a sequence of V_i's or W_j's to X. The same change $p(x,T)$ for any $x \in D$. This gives (i). ∎

Note that the lemma implies every outer blossom B has a root vertex in T. This root is used below. Although we do not need this fact, the root of B is its "base vertex" [GMG].

We incorporate the above algorithm to construct T into a search of Edmonds' algorithm. Using the algorithm of Section 3 the time for *add_leaf* is $O(1)$, so the time for a search increases by only $O(n)$. This follows since using a data structure such as [GMG, GT89] mentioned above, the

algorithm finds the vertices to be added to T in time linear in their number.

Even using the extra information provided by tree T, the blossom merging problem seems difficult. We will simplify the problem: instead of processing general edges vw (in *make_edge* operations) we will work with back edges. By property (*iii*), when Edmonds' algorithm does a blossom step for an edge vw it has precisely the same effect as doing blossom steps for the two edges va and wa. We will replace vw by these two back edges.

In summary we implement Edmonds' algorithm as follows. The search algorithm constructs tree T using *add_leaf* operations. (As part of this operation, each vertex y records its depth $d(y)$ in T.) In addition when the search discovers an edge vw joining two outer vertices, it performs $nca(v, w)$ to find the nearest common ancestor a in T. Then it executes the operations $make\_edge(va, t)$ and $make\_edge(wa, t)$. Hence all edges in the blossom merging problem are back edges.

The rest of the implementation is a solution to the *tree-blossom merging problem.* It is defined on a tree T that is constructed incrementally. There are three operations, *make_edge*, *merge* and *find_min*, defined as in the blossom merging problem with the restriction that all edges of $\mathcal{E}$, i.e., the arguments to *make_edge*, are back edges. (There is no *make_blossom* operation, since *add_leaf* adds vertices.) Note that T contains vertices that are not outer, but this is inconsequential.

Let us prove that this implementation of Edmonds' algorithm is correct. The main observation is that an edge vw in the given graph joins two distinct outer blossoms if and only if at least one of its corresponding edges va, wa in $\mathcal{E}$ joins two distinct blossoms. This observation holds by Lemma 2.1. The observation shows that *find_min* returns (a back edge corresponding to) the desired minimum edge. Thus the implementation is correct.

Section 3 shows that the total time for $O(n)$ *add_leaf* operations and $O(m)$ *nca* operations is $O(m + n)$. The next two subsections solve the tree-blossom merging problem in the desired time.

2.2. The simple merging algorithm.

This section gives an algorithm for tree-blossom merging that uses $O(m + n \log^2 n)$ time and $O(m + n \log n)$ space. This implements a search of Edmonds' algorithm in the same resources.

The fact that tree T is constructed incrementally is not central to our merging algorithms. (Of course it is important in Section 3.) The algorithms use only a simple representation of T: each vertex x in T records its parent and its depth $d(x)$ in T. It is a simple matter to record these values in total time $O(n)$: The operation $add\_leaf(x, y)$ computes the values for y from those of x.

For a set of edges S, define $smallest(S)$ as an edge in S with smallest possible cost. If

several edges tie for smallest cost, $smallest(S)$ can be chosen from amongst them arbitrarily (so $smallest(S)$ always denotes one edge). If U and U' are disjoint sets of vertices then $smallest(U, U')$ is defined similarly, referring to the edges of $\mathcal{E}$ from U to U'. For a vertex v, $smallest(U, v)$ denotes $smallest(U, \{v\})$. Note that if B is a blossom, for any vertex v it suffices for the algorithm to retain an edge $smallest(B, v)$ – all other edges from B to v can be discarded, since they need never be returned as the value of *find_min*.

The data structure has three main parts: a priority queue $\mathcal{F}_0$, a data structure for edges and a data structure for blossoms. We describe each in turn. Several Fibonacci heaps are used. We make the convention of using an edge e to represent a key in these heaps. Strictly speaking the key is the cost of e, but e is stored in the heap anyway; no confusion will result.

The priority queue $\mathcal{F}_0$ keeps track of the smallest blossom step edge incident to each blossom. More precisely $\mathcal{F}_0$ is a Fibonacci heap, with an entry for each blossom B. The entry for B is $smallest(B, O - B)$. Thus a *find_min* operation in our data structure amounts to a Fibonacci heap *find_min* in $\mathcal{F}_0$.

To define the edge data structure, say that the *rank* of an edge $vw \in \mathcal{E}$ equals $\lfloor \log |d(v) - d(w)| \rfloor$. Thus the rank of an edge is between zero and $\lfloor \log (n-1) \rfloor$. The edge data structure is a partition of $\mathcal{E}$ into sets called packets. The *packet of rank r for vertex v* contains all edges $vw \in \mathcal{E}$ with rank r and $d(v) > d(w)$. A packet P is implemented as a linked list of edges; it also records $smallest(P)$. Each vertex v has an array $packet_v[0..\lfloor \log n \rfloor]$ of pointers to its packets, i.e., $packet_v[r]$ points to the packet of rank r for v. Observe that a given edge can be added to the appropriate packet in $O(1)$ time. (This notion of packet is similar in spirit, but not detail, to the data structure of the same name in [GGS, GGST].)

To *open a packet* P means to transfer the edges of P into the rest of the data structure. This destroys P, i.e., a packet is opened only once. It is useful to be able to restrict the *smallest* function to edges coming from opened packets – for this we use the function *o-smallest*. For example, $o\text{-}smallest(U, v)$ denotes the smallest edge of $\mathcal{E}$ from U to v whose packet has been opened.

The third part of the data structure is for blossoms. A blossom with s nodes has *rank* $\lfloor \log s \rfloor$. A *minimal blossom of rank r* is a blossom of rank r resulting from the *merge* of two blossoms of rank less than r. Clearly the minimal blossoms of a given rank r are pairwise vertex disjoint. Any blossom B of rank r contains a unique minimal blossom B_0 of rank r. The *minimal root* of B is the root of B_0 in T. It exists because of Lemma 2.1.

The data structure for a blossom B is as follows. There are several bookkeeping items: The rank r and minimal root b are recorded. There is also a representation of the partition of O into

blossoms. A data structure for set merging [T83] can be used: The blossom merging operation $merge(A, B)$ executes a set merging operation $union(A, B)$ to construct the new blossom; for any vertex v, the set merging operation $find(v)$ gives the blossom currently containing v.

For simplicity the algorithms of Sections 2.2 – 2.3 omit the obvious details associated with this bookkeeping. We can also ignore the time and space. For suppose we use a simple set merging algorithm that does one *find* in $O(1)$ time, all *unions* in $O(n \log n)$ time, and uses $O(n)$ space [e.g., AHU]. Clearly the space and the time for *unions* are within the desired bounds for Edmonds' algorithm; the time for *finds* can be associated with other operations. Hence we shall ignore this bookkeeping.

The heart of the blossom data structure is a Fibonacci heap $\mathcal{F}(B)$ for each blossom B. Every vertex of B has all its packets of rank at most r opened. Heap $\mathcal{F}(B)$ has two types of entries:

(i) Each of the first $3 \cdot 2^r$ ancestors v of b (the minimal root of B) has an entry. The heap key of v is $o\text{-}smallest(B, v)$ if $v \notin B$; it is ∞ if $v \in B$.

(ii) Each unopened packet P of a vertex of B has an entry. The key of P is $smallest(P)$.

The entry for B in heap $\mathcal{F}_0$ is the minimum of $\mathcal{F}(B)$.

A given vertex v may simultaneously have entries in a large number of heaps $\mathcal{F}(B)$. (This occurs when v is an ancestor of a large number of vertices minimal roots b.) The algorithm must be able to find the entry for a given v in a given $\mathcal{F}(B)$ in $O(1)$ time. To do this, each blossom B of rank r has an array $vertex_B[1 \,..\, 3 \cdot 2^r]$ of pointers to the vertex entries in $\mathcal{F}(B)$ – specifically $vertex_B[i]$ gives the entry for the ancestor of b at depth $d(b) - i$. Thus given an ancestor v the corresponding entry in $\mathcal{F}(B)$ is $vertex_B[d(b) - d(v)]$.

The correctness of the data structure amounts to the following result, which shows that $\mathcal{F}_0$ contains the correct information for each blossom.

Lemma 2.2. At any time for any blossom B, the minimum entry in $\mathcal{F}(B)$ equals $smallest(B, O - B)$.

Proof. Let B have rank r. First observe that a blossom containing both ends of an edge of rank r has rank at least r. This follows from Lemma 2.1. It justifies the fact that the data structure for B does not open any packet P of rank more than r: if a blossom step is done for $smallest(P)$, this will increase the rank of the blossom so that a new blossom data structure gets initialized (in the new data structure P gets opened).

Next observe that if a packet containing an edge vw, $d(v) > d(w)$, has been opened then w has an entry in $\mathcal{F}(B)$. This follows from two inequalities: Since the packet is opened its rank is at most r, whence $d(w) > d(v) - 2^{r+1}$. Since the rank of B is r, less than 2^r vertices have been to its minimal rank r blossom. Thus $d(v) > d(b) - 2^r$. Transitivity implies $d(w) > d(b) - 3 \cdot 2^r$, i.e., w has an entry in $\mathcal{F}(B)$. Clearly it suffices that the entry for w maintains *o-smallest*(B, w) – an edge in an opened packet that leads to w but is not *o-smallest*(B, w) is irrelevant.

These two observations show that $\mathcal{F}(B)$ contains all possible candidates for *smallest*$(B, O-B)$. Furthermore each finite entry in $\mathcal{F}(B)$ corresponds to an edge from B to $O - B$. Thus the data structure is correct. ∎

The four blossom merging operations are implemented so that they maintain the defining properties of the data structure. A *make_blossom* operation simply updates the representation of the partition of O into blossoms. As already noted a *find_min* operation is trivial given the heap $\mathcal{F}_0$. Now we describe the two other operations.

Consider *make_edge*(vw, t), where $d(v) > d(w)$ and v is in blossom B. Compute the rank of vw as $r = \lfloor \log(d(v) - d(w)) \rfloor$. If r is at most the rank of B then vw belongs to an opened packet. Update the key for w in $\mathcal{F}(B)$ by possibly doing a Fibonacci heap *decrease_key* operation (find the heap entry for w using $vertex_B[d(b) - d(w)]$). Otherwise use $packet_v[r]$ to find the unopened packet P. Add vw to P, updating *smallest*(P) and the corresponding Fibonacci heap entry if necessary. Finally if the minimum of $\mathcal{F}(B)$ decreases do a *decrease_key* operation for B's entry in $\mathcal{F}_0$.

Consider *merge*(A, B). Let C denote the blossom resulting from the merge. Suppose first that C has the same rank r as one of its constituents, say B; clearly A has rank less than r. The algorithm forms the data structure for C by reading the information for A into the data structure for B. More precisely the following is done. The data structure for B is now associated with C, e.g., $\mathcal{F}(B)$ is now $\mathcal{F}(C)$. Entries for vertices in $\mathcal{F}(A)$ are used to update entries for vertices in $\mathcal{F}(C)$ by possibly doing *decrease_key* operations. Packets of A with rank at most r are opened and their edges are used to update entries for vertices in $\mathcal{F}(C)$. Each unopened packet of A is inserted into $\mathcal{F}(C)$. The Fibonacci heap operation *delete_min* is done in $\mathcal{F}(C)$ as long as the minimum entry is a vertex in C. The entries for A and B in $\mathcal{F}_0$ are deleted and replaced by the new heap minimum in $\mathcal{F}(C)$.

The second case for *merge*(A, B) is when C has a larger rank r than either constituent. (Thus C is a minimal blossom of rank r.) The algorithm begins by initializing a new data structure for C. The information for both A and B is read into this data structure, following the procedure of

the first case.

Correctness of these algorithms is obvious – they maintain the invariants of the data structure. We show the entire blossom merging algorithm achieves the desired resource bounds, $O(m+n\log^2 n)$ time and $O(m+n\log n)$ space. To do this we make a preliminary observation: Over the entire algorithm, the total number of vertex entries in all distinct heaps $\mathcal{F}(B)$ is $O(n\log n)$. To show this note that the distinct heaps correspond to the heaps of minimal blossoms B of each rank r. Such a blossom B contains at least 2^r vertices and $\mathcal{F}(B)$ has $O(2^r)$ vertex entries. As already noted the minimal blossoms of rank r are pairwise vertex disjoint. Thus the distinct heaps for rank r have $O(n)$ vertex entries. This implies the desired total of $O(n\log n)$ vertex entries.

Now we show the space bound. The $packet_v$ tables use $O(n\log n)$ space. The heaps $\mathcal{F}(B)$ use $O(n\log n)$ space, since at any given time there are $O(n)$ heap entries for vertices and $O(n\log n)$ entries for packets. (For vertices, note that at any time the existing blossoms are pairwise vertex disjoint.) In addition the $vertex_B$ tables are stored. At any time the existing heaps $\mathcal{F}(B)$ have a total of $O(n)$ entries in these tables. This does not necessarily imply a space bound of $O(n)$, since contiguous space must be allocated dynamically as blossoms grow. Instead of introducing a space allocation strategy we simply note that the total size of all tables ever existing is $O(n\log n)$, by the preliminary observation. It is clear that the remaining space is $O(m)$, so the desired space bound follows.

Next we prove the time bound. An operation *make_blossom*(A) uses $O(|A|)$ time. A *find_min* operation uses $O(1)$ time and a *make_edge* operation uses $O(1)$ amortized time. This amounts to $O(m)$ time total. It remains only to estimate the time for all *merge* operations.

As observed above there are $O(n\log n)$ vertex entries in all distinct Fibonacci heaps. Each such entry can be deleted, using $O(n\log^2 n)$ time total. Similarly each vertex entry can cause a *decrease_key* when it is transferred to a larger heap, using $O(n\log n)$ time total. Each packet gets inserted in $O(\log n)$ distinct heaps $\mathcal{F}(B)$, using $O(n\log^2 n)$ time total. It gets deleted from a heap only once, right before that heap is discarded, using $O(n\log n)$ time. Each edge uses $O(1)$ time when its packet is opened. These contributions dominate the total time. We conclude that the simple merging algorithm works as desired.

2.3. The refined merging algorithm.

This section implements Edmonds' algorithm with the desired efficiency, $O(m+n\log n)$ time and $O(m)$ space. It does this by presenting an algorithm for tree-blossom merging that uses the same resources. (We still assume the tree operations of Section 3.)

We start by a describing useful operation, also used in [GGS]. Consider a blossom B and a set A of consecutive ancestors of B. Let S be a set of edges from opened packets, each joining A and B, containing the edge *o-smallest*(B,a) for each $a \in A$. To *clean* S means to discard all edges of S except the edges *o-smallest*(B,a), $a \in A$.

A cleaning operation can be done in time $O(|A|)$ plus $O(1)$ per edge discarded, and space $O(|A|)$. To do this suppose A consists of the ancestors of B with depths in the range $[\ell..h]$. Read the edges of S into an array $A[\ell..h]$, where $A[i]$ keeps track of the smallest edge from B to the depth i ancestor.

Now we describe the data structure for the refined algorithm. It is based on this classification: A blossom is *small* if it has less than $\lfloor \log n \rfloor$ vertices. Otherwise it is *big*.

Consider a small blossom B. Let S denote the set of edges of $\mathcal{E}$ that are incident to B. The edges of S are stored in two data structures associated with B, a packet $\mathcal{P}(B)$ and a list $\mathcal{L}(B)$. $\mathcal{P}(B)$ uses the data structure of Section 2.2 for packets (i.e., a linked list), but in addition to having a pointer to the first element $\mathcal{P}(B)$ has a pointer to the last element. $\mathcal{P}(B)$ contains all edges of S that have rank at least $\lfloor \log n \rfloor$. It is natural to think of $\mathcal{L}(B)$ as containing edges of a fictitious packet that has been opened. Thus the function *o-smallest* takes into account all edges ever in $\mathcal{L}(B)$. $\mathcal{L}(B)$ contains edges of S with rank less than $\lfloor \log n \rfloor$ joining B to $O - B$; for each vertex a adjacent to B along such an edge, $\mathcal{L}(B)$ contains *o-smallest*(B,a) and possibly other edges incident to a. The entry for B in $\mathcal{F}_0$ is *smallest*$(\mathcal{P}(B) \cup \mathcal{L}(B))$.

This data structure is correct – the entry for B in $\mathcal{F}_0$ is the same as the desired entry *smallest*$(B, O - B)$. To show this observe that $\mathcal{P}(B) \cup \mathcal{L}(B)$ contains *smallest*$(B, O - B)$ by the above discussion. Furthermore $\mathcal{P}(B) \cup \mathcal{L}(B)$ contains only edges leading out of B. This follows because an edge of $\mathcal{P}(B)$ has rank larger than B.

Observe also that $\mathcal{L}(B)$ contains edges leading to $O(\log n)$ distinct ancestors of B.

Now we give the data structure for big blossoms. It uses packets and heaps. We discuss these in turn.

For packets, consider a minimal big blossom B, i.e., B has at least $\lfloor \log n \rfloor$ vertices but neither of its constituents does. B has $\lfloor \log n \rfloor + 1$ packets, just like a vertex in the simple algorithm. As in the simple algorithm the packet of rank r contains edges of rank r. (A difference is that now edges in a packet of rank r can lead to vertices differing in depth by more than 2^{r+1}.) The packets for B are maintained throughout the algorithm, even when B gets absorbed into larger blossoms C. (In fact after B gets absorbed, edges may get added to its packets. These edges will be incident to C but need not be incident to B.) Note that there are $O(n)$ such packets total, since there are

$O(n/\log n)$ such blossoms B, each with $O(\log n)$ packets.

For heaps, consider a big blossom B with rank r (thus $r \geq \log^{(2)} n$). Let b be its minimal root. B has a Fibonacci heap $\mathcal{F}(B)$, that is the same as in the simple algorithm (see (i) - (ii) of the discussion of heaps in Section 2.2) with one change, in how the ancestors of b are processed. The first $3 \cdot 2^r$ ancestors of b are partitioned into *groups* of $\lfloor \log n \rfloor$ consecutive vertices. Groups are treated like vertices in the simple algorithm. Specifically, $\mathcal{F}(B)$ has one entry for each group G. The key for G is *o-smallest*$(B, G - B)$. (This is infinite if $G \subseteq B$.) Each group G has a list $\mathcal{L}(G)$ of edges of $\mathcal{E}$ from B to G; $\mathcal{L}(G)$ contains *o-smallest*(B, w) for each vertex $w \in G$, if it exists, and possibly other edges incident to w. Finally there is an array $group_B[1 \,..\, \lceil \frac{3 \cdot 2^r}{\lfloor \log n \rfloor} \rceil]$ of pointers to the groups ($group_B[i]$ points to the group of ancestors of b with depths in the range $(d(b) - (i-1)\lfloor \log n \rfloor \,..\, d(b) - i\lfloor \log n \rfloor]$). Note that over the entire algorithm there are $O(n)$ groups total. This follows since the minimal blossoms for rank r collectively contain at most n vertices. Since these blossoms are all big they collectively contain $O(n/\log n)$ groups.

The algorithm maintains the invariant that the packets for all blossoms included in a big blossom B collectively contain all edges incident to B. (Thus when a vertex enters a big blossom for the first time its edges are placed in packets of the appropriate rank. This is true even if the packet has been opened, by convention.) The entry for a big blossom B in heap $\mathcal{F}_0$ is the minimum entry in $\mathcal{F}(B)$. The same reasoning as in the simple algorithm shows this entry is correct.

Now we sketch the algorithms using these data structures. We also show that each algorithm achieves the desired time bound, and uses permissible auxiliary space. The operations *make_blossom* and *find_min* are straightforward, as in the simple algorithm.

Consider *make_edge*(vw, t), where $d(v) > d(w)$ and v is in blossom B. The rank r of vw is computed. If B is small then vw is added to $\mathcal{P}(B)$ or $\mathcal{L}(B)$, depending on the rank r. The entry for B in $\mathcal{F}_0$ is updated if appropriate. Suppose B is big. If r is greater than the rank of B then vw belongs in an unopened packet; an unopened packet of rank r for some blossom in B is chosen arbitrarily and vw is added to it. Otherwise (when r is at most the rank of B) the group G for w is found (using $group_B[\lceil \frac{d(b) - d(w)}{\lfloor \log n \rfloor} \rceil]$) and vw is added to $\mathcal{L}(G)$. In both cases, *decrease_key* is done in the heaps $\mathcal{F}(B)$ and $\mathcal{F}_0$ if appropriate.

It is clear that *make_edge* uses $O(1)$ (amortized) time. This depends on the fact that lists $\mathcal{L}(B)$ and $\mathcal{L}(G)$ do not get cleaned when an edge is added.

Next consider an operation *merge*(A, B). Let C denote the blossom resulting from the merge. We consider several cases. For each case we state the algorithm. We also show the time for all merges of that case is $O(m + n \log n)$. To do this we use the following accounting scheme: We

charge $O(\log n)$ to each *merge* operation, $O(1)$ to each vertex or packet whose new blossom C has larger rank, $O(\log n)$ to a group when it is discarded, and one or two charges of $O(1)$ to each edge (one of the charges is when the edge is discarded). It is easy to see that this implies the desired time bound.

Suppose C is small. Clearly A and B are small. Packet $\mathcal{P}(C)$ is formed by concatenating $\mathcal{P}(A)$ and $\mathcal{P}(B)$. List $\mathcal{L}(C)$ is formed by cleaning $\mathcal{L}(A) \cup \mathcal{L}(B)$. The value $smallest(\mathcal{P}(B) \cup \mathcal{L}(B))$ is determined and $\mathcal{F}_0$ is updated, as in the simple algorithm.

To estimate the time note that cleaning uses time $O(\log n)$ plus $O(1)$ per edge discarded. This follows since as already observed, the the edges of $\mathcal{L}(A) \cup \mathcal{L}(B)$ lead to $O(\log n)$ distinct ancestors of C. The desired time bound follows. Only $O(\log n)$ auxiliary space is used in the cleaning.

Next suppose C is big. We first assume that B and C have the same rank. As in the simple algorithm, *merge* reads the information for A into the data structure for B. The details depend on whether A is small or big. We consider each case in turn.

Suppose A is small. First $\mathcal{L}(A)$ is cleaned. Then *make_edge* is done for every edge in $\mathcal{P}(A) \cup \mathcal{L}(A)$. (Recall that heap $\mathcal{F}(C)$ is what was previously $\mathcal{F}(B)$.) Next the following procedure is executed as long as the minimum entry of $\mathcal{F}(C)$ is for a vertex v in C: The Fibonacci heap operation *delete_min* removes the entry for v and its group, say group G. If $G \subseteq C$ then the edges in $\mathcal{L}(G)$ are discarded and nothing else is done. Otherwise $\mathcal{L}(G)$ is cleaned and a new entry for G is added to $\mathcal{F}(C)$. When this loop finishes, the minimum entry of $\mathcal{F}(C)$ is for a vertex not in C (or it is ∞). The entries for A and B in $\mathcal{F}_0$ are deleted and replaced by the new heap minimum of $\mathcal{F}(C)$.

To estimate the time, observe that an edge in $\mathcal{P}(A)$ is charged $O(1)$ time for its *make_edge* operation, but this occurs only once (since it is now associated with a big blossom). Since $\mathcal{L}(A)$ is cleaned its *make_edge* operations use $O(\log n)$ time, charged to the *merge* operation. Next note that at most one group H contains vertices from both A and B (from Lemma 2.1). If a *delete_min* operation is done for a group $G \neq H$ then $G \subseteq B$. Thus G gets discarded, so it can be charged $O(\log n)$ time for the *delete_min*. Now consider the *delete_min* for group H. Its list $\mathcal{L}(H)$ is cleaned in time $O(\log n)$ plus $O(1)$ per edge discarded (since a group has $\lfloor \log n \rfloor$ vertices). The *merge* operation is charged $O(\log n)$ for the cleaning overhead and the *delete_min*. It is easy to see that the desired time bound follows.

Next suppose A is big (and as already assumed, C is big, B and C have the same rank.) The strategy is still to read the information for A into the data structure for B. Most details are the same as the previous case. We need only describe how a group G of A is transferred to C: The list

$\mathcal{L}(G)$ is cleaned, and *make_edge* is done for each entry in the cleaned list.

The time is analyzed similar to the previous case, with an additional remark to analyze the above processing of a group G: The overhead for cleaning each list $\mathcal{L}(G)$ is $O(\log n)$. After cleaning $\mathcal{L}(G)$ has at most $\lfloor \log n \rfloor$ entries; they use $O(\log n)$ time for *make_edge* operations. G can be uniquely associated with $\Omega(\log n)$ vertices of A. These vertices are now in a blossom of larger rank. Charging each one $O(1)$ accounts for the time.

It remains only to consider the case of C big and having larger rank than A or B. The algorithm begins by initializing a new data structure for C. The information for both A and B is read into this data structure, following the procedure of previous cases.

We have proved the desired time bound. Now we show the space bound. The big blossoms have $O(n)$ packets total and $O(n)$ groups total. Thus the heaps $\mathcal{F}(B)$ use $O(n)$ space, as do the arrays $packet_B$ and $group_B$. Additional space is $O(m)$.

Theorem 2.1. A search in Edmonds' algorithm can be implemented in $O(m + n \log n)$ time. The weighted matching problem can be solved in $O(n(m + n \log n))$ time. In both cases the space is $O(m)$. ∎

We conclude this section by mentioning some problems that can be solved by a search of Edmonds' algorithm. The *matching update problem* is to maintain a maximum-weight matching in a graph that can be repeatedly modified at an arbitrarily chosen vertex. More precisely we have a graph with edge weights, that is initially empty. An operation can add a new vertex together with arbitrary incident edges having arbitrary weights, or delete a vertex. (Doing both of these amounts to modifying the edges incident to a given vertex arbitrarily.) A maximum-weight matching must be output after each operation. This problem can be solved by doing one or two searches of Edmonds' algorithm for each update operation (plus an additional $O(m)$ processing). Thus the update problem can be solved in $O(m + n \log n)$ time per update operation [BD, CM, G85b, W]. The same result holds for variants of the problem like maintaining a maximum-weight maximum cardinality matching. The matching update problem has been used in an algorithm for the set partitioning problem [NW] and in a mass transit crew scheduling system [BD].

Our algorithm extends to an efficient implementation of a search for the degree-constrained subgraph problem [L]. One simple way to see this is to use the sparse substitute technique of [G83], which reduces a search of the degree-constrained subgraph algorithm to a search of the matching algorithm. An application of this is the all-pairs shortest path problem on an undirected graph with no negative cycles [L]. [G83] shows that this problem can be solved by doing a search for each

source, implying resource bounds of $O(n(m + n\log n))$ time and $O(m)$ space. (The space is $O(n^2)$ space if the distance matrix must be stored).

3. Nearest common ancestors.

This section presents algorithms for two nearest common ancestor problems. It starts with the incremental-tree nearest common ancestor problem. Section 3.1 gives a one-level algorithm. Section 3.2 refines this to a multi-level algorithm having optimal run time, $O(m + n)$. Section 3.3 solves the nearest common ancestors with linking problem in time $O(m\alpha(m, n) + n)$.

We use the following tree terminology, in addition to the terms introduced in Section 1: For a node v in a tree T, let T_v denote the subtree rooted at v.

The algorithms presented reduce an *nca* query to evaluations of the following more informative function on auxiliary trees. Fix a tree and consider nodes x, y. Let $a = nca(x, y)$. For $z = x, y$, let a_z be the ancestor of z immediately preceding a; if $a = z$ then take $a_z = a$. Define $ca(x, y)$, the *characteristic ancestors of x and y*, as the ordered triplet (a, a_x, a_y). The algorithms for *nca* compute *ca*.

3.1. One-level algorithms.

This section starts with an algorithm to find nearest common ancestors on a tree that is given in advance. Then it extends the algorithm, first to process *add_leaf* operations and then to solve the general incremental-tree problem. The algorithms are simple in that they use only one level. The incremental-tree algorithm runs in $O(m + n\log^2 n)$ time.

As in [HT] the main auxiliary tree used is the compressed tree. Let us review the basic definitions [HT, T79]. Let T be a tree with root $r(T)$. The *size* $s(v)$ of a node v is the number of its descendants. (As usual a node is a descendant of itself.) A child w of v is *light* if $s(v) \geq 2s(w)$; otherwise it is *heavy*. Deleting each edge from a light child to its parent leaves a set of disjoint paths (each with nonnegative length) called the *heavy paths of T*. A node is an *apex* if it is not a heavy child; equivalently it is the highest node on some heavy path.

Consider any partition of $V(T)$ into a family $\mathcal{P}$ of disjoint paths of nonnegative length. Call the highest node on each path its apex. The *compressed tree for T and $\mathcal{P}$*, $C(T, \mathcal{P})$, has nodes $V(T)$ and root $r(T)$; the parent of a node $v \neq r(T)$ is the first proper ancestor of v that is an apex. When $\mathcal{P}$ consists of the heavy paths of T we call this the *compressed tree (for T)*, denoted $C(T)$.

For any tree T, the height of the compressed tree $C = C(T)$ is at most $\lfloor \log n \rfloor$. This follows from the fact that in C, the parent v of node w has $s_C(v) \geq 2s_C(w)$.

We start by showing how to compute $nca_C(x, y)$ in $O(1)$ time, when C is the compressed tree for some arbitrary given tree. The method of [HT] embeds C in a complete binary tree B; *ncas* in B are calculated using the binary expansion of the inorder numbers of the nodes. [SV] uses a similar approach. Our method is different and seems to give simpler algorithms (see Section 3.2).

Fix an arbitrary tree C. (For generality we do not assume that C is a compressed tree at this point.) In what follows all tree functions refer to C (e.g., s denotes s_C). Generalizing compressed trees, assume there is a constant $\beta > 1$ such that for any node v with child w,

$$s(v) \geq \beta s(w). \tag{1}$$

Let $\mathbf{N}$ be the set of natural numbers. Choose positive integers e and c that satisfy

$$\frac{2}{\beta^{e-1} - 1} \leq c - 2 \leq \beta^e; \tag{2}$$

for instance for $\beta = 2$, $e = 2$ and $c = 4$. A *fat preorder numbering of* C is a function $f : V(C) \to \mathbf{N}$ for which there are functions $g, f^*, g^* : V(C) \to \mathbf{N}$ such that for any node v,

(i) the descendants of v are precisely the nodes w having $f(w) \in [f(v)..g(v))$;
(ii) there are no values $f(u)$ in $[f^*(v)..f(v)) \cup [g(v)..g^*(v))$;
(iii) $g^*(v) - f^*(v) = cs(v)^e$ and $f(v) - f^*(v),\ g^*(v) - g(v) \geq s(v)^e$.

Note that property (i) is equivalent to f being a preorder numbering.

Given a fat preorder numbering, the following algorithm returns $nca_C(x, y)$: Let a be the first ancestor of x that has $(c-2)s(a)^e \geq |f(x) - f(y)|$. If a is an ancestor of y (i.e., $f(a) \leq f(y) \leq g(a)$) then return a else return $p(a)$ (the parent of a).

Lemma 3.1. The nca_C algorithm is correct.

Proof. First observe that $nca_C(x, y)$ is an ancestor of node a. For consider any common ancestor b of x and y. By (i) the interval $[f(b)..g(b))$ contains both $f(x)$ and $f(y)$, i.e., its length is at least $|f(x) - f(y)|$. By (iii) its length is at most $(c-2)s(b)^e$. Thus $(c-2)s(b)^e \geq |f(x) - f(y)|$. This implies the desired conclusion.

We show that $p(a)$ is a common ancestor. We need only prove that y is a descendant of $p(a)$. By (i) – (iii) a nondescendant of $p(a)$ and a descendant of $p(a)$ differ in number by more than $s(p(a))^e$. However using (1) and the right inequality of (2), $s(p(a))^e \geq \beta^e s(a)^e \geq (c-2)s(a)^e \geq |f(x) - f(y)|$. Thus y is a descendant of $p(a)$.

The two observations imply that $nca(x,y)$ equals a or $p(a)$. ∎

To implement the algorithm and also to compute characteristic ancestors, we use the following data structure. Each vertex x stores an *ancestor table*, $ancestor_x[0\ ..\ \lfloor e(1+\log_\beta n)\rfloor]$. The entry $ancestor_x[i]$ is the last ancestor b of x that has $(c-2)s(b)^e < \beta^i$; if no such ancestor exists the entry is $\emptyset$.

We show how to compute the characteristic ancestors in $O(1)$ time. It suffices to show how to find $nca_C(x,y)$ plus the characteristic ancestor a_x. For this it suffices to find the ancestor a used in the nca_C algorithm, plus the ancestor of x that precedes it. Let $i = \lfloor \log_\beta |f(x) - f(y)| \rfloor$ and $v = ancestor_x[i]$. Assume $v \neq \emptyset$ (the reader can easily supply the details for the degenerate case $v = \emptyset$). Thus $w = p(v)$ is the first ancestor of x that has $(c-2)s(w)^e \geq \beta^i$. Using (1) and $e \geq 1$ shows $(c-2)s(p(w))^e \geq \beta^{i+1}$. Thus the desired a is either w or $p(w)$. The ancestor preceding a is v or w. The time is clearly $O(1)$. Thus we have shown how to compute $ca_C(x,y)$ in $O(1)$ time.

Next we show that a fat preorder numbering of tree C exists and can be constructed in $O(n)$ time. We give recursive algorithm to number C in fat preorder. It traverses C top-down; when visiting a node u, u will have already been assigned an interval $[f^*(u)..g^*(u))$ with $g^*(u) - f^*(u) = cs(u)^e$. Initially assign root $r(C)$ the interval $[0..cn^e)$. To visit u, assign $f(u) \leftarrow f^*(u) + s(u)^e$, $g(u) \leftarrow g^*(u) - s(u)^e$; then assign intervals to the children of u, starting at $f(v)+1$, as follows: For each child v of u, assign an interval of length $cs(v)^e$ to v and then visit v.

Lemma 3.2. The fat preorder numbering algorithm is correct.

Proof. It is clear that the algorithm achieves properties (i) – (iii) of fat preorder. We must show that the intervals assigned by u all fit into the interval $[f^*(u)..g^*(u))$ given to u. For u a leaf this holds since $c \geq 3$ (by (2)). Assume u is a nonleaf, and let U denote the set of children of u. Starting with the relation $s(u) = 1 + \sum\{s(v) \mid v \text{ a child of } u\}$, multiply by $s(u)^{e-1}$ and use (1) and $s(u) \geq \beta$ to obtain $s(u)^e \geq \beta^{e-1} + \beta^{e-1} \sum\{s(v)^e \mid v \in U\}$. This implies $cs(u)^e/\beta^{e-1} \geq 1 + \sum\{cs(v)^e \mid v \in U\}$. The right-hand side is the total size of intervals assigned in $[f(u)..g(u))$ (the term 1 accounts for the number assigned to u, $f(u)$). Since $[f(u)..g(u))$ has length $(c-2)s(u)^e$ it suffices to have $c - 2 \geq c/\beta^{e-1}$. This is equivalent to the left inequality of (2). ∎

The last important step in the *nca* algorithm is a procedure, due to [HT], relating characteristic ancestors in T and its compressed tree $C(T)$. To state it let C be the compressed tree for T and an arbitrary set of paths $\mathcal{P}$. Suppose the characteristic ancestors $ca_C(x,y) = (a, a_x, a_y)$ are known.

The definition of C implies that $nca_T(x,y)$ is the first common ancestor of x and y on the path of $\mathcal{P}$ with apex a. For $z = x, y$ let b_z denote the first ancestor of a_z on the path with apex a, i.e., b_z is a_z or $p_T(a_z)$. Then $nca_T(x,y)$ is b_x or b_y, whichever is closer to a. It is easy to extend this to compute $ca_T(x,y)$.

Putting these pieces together gives our algorithm for *nca* queries on a static tree. Let us summarize the algorithm. A preprocessing step computes the compressed tree $C = C(T)$. It is numbered in fat preorder; in addition the order of nodes in each heavy path is recorded. The ancestor tables for C are constructed. The query algorithm computes characteristic ancestors, first finding $ca_C(x,y)$ and using that to find $ca_T(x,y)$.

Lemma 3.3. A tree T with n nodes can be preprocessed using $O(n \log n)$ time and space so that an *ca* query can be answered in $O(1)$ time. ∎

Note that the preprocessing time and the space are both $O(n)$ except for the resources needed to compute and store the ancestor tables.

Next we consider trees that grow by *add_leaf* operations. The algorithm is based on a dynamic version D of the compressed tree. D is the compressed tree for T and a time-varying set of paths $\mathcal{P}$. Note that $\mathcal{P}$ distinguishes a subset of the vertices of D as apexes. The algorithm to construct D is based on this operation: Let v be an apex. Thus $V(D_v) = V(T_v)$. To *recompress* v means to replace D_v in D by $C(T_v)$ ($C(T_v)$ is defined using the heavy paths of T_v). Any node of T_v gets *reorganized* in this operation.

Fix a constant $\alpha > 1$. For a node w, let $s(w)$ denote its current size in D and let $s_0(w)$ denote its size in D when it was last reorganized. For example if a node is not an apex then both values equal one. D is maintained to always have $s(w) < \alpha s_0(w)$ for every node w.

The data structure maintains the values $s(w)$ and $s_0(w)$. It also marks the apexes. Tree D is represented by parent pointers and tree T is represented by children lists (i.e., each node has a list of its children).

Now we give the algorithms for *add_leaf* and *ca*. To do *add_leaf*(x,y), add y to the list of children of x in T. Make y a singleton path of $\mathcal{P}$ by marking it an apex and setting $p_D(y)$ to x if x is an apex, else $p_D(x)$. Next increase $s(a)$ by one for each ancestor a of y in D. Let v be the last ancestor v of y that now has $s(v) \geq \alpha s_0(v)$ (this condition holds for $v = y$ by convention). Recompress v and halt.

There is one more detail of the recompression operation. The algorithm maintains a fat preorder numbering of D. The fat preorder satisfies the defining properties (i) – (iii) and two

additional properties: Set $\beta = 1 + \frac{1}{2\alpha - 1}$ and replace inequality (2) by

$$c\frac{(\alpha - 1/2)^e + (1/2)^e}{1 - (1/\alpha)^e} \leq c - 2 \leq \beta^e; \tag{3}$$

for instance $\alpha = 5/4$, $e = 4$, $c = 6$. Let $ex(v)$ be the largest value $g^*(z)$ for a descendant z of v in D. The *expansion interval* for v is $[ex(v)..g(v))$. When the *add_leaf* algorithm recompresses a node v with parent $u = p_D(v)$, it uses the fat preorder numbering algorithm to assign new numbers to the nodes of T_v, in the interval $[ex(u)..ex(u) + cs(v)^e)$. This decreases the size of u's expansion interval. Also the old interval for v, $[f^*(v)..g^*(v))$, is in effect discarded. (If v is the root of D there is no need for an expansion interval – all nodes receive new numbers in the interval $[0..cs(v)^e)$.) The last step of the *add_leaf* algorithm is to construct new ancestor tables for all nodes of T_v.

The *ca* algorithm is the same as for the static case.

Before proving this algorithm correct note two differences from the static case: First, in general $D \neq C(T)$ – a child v of node u in T may be an apex in D, but it may have grown to become u's heavy child. Second, the algorithm for $ca(x, y)$ uses old information – the ancestor table of x may have been constructed before y had its current preorder number.

Lemma 3.4. The *add_leaf* and *ca* algorithms are correct.

Proof. First observe that at any time when v is a child of u in D, $s(u) \geq \beta s(v)$. To show this suppose that when u was last reorganized, $s(u) = s_u$ and $s(v) = s_v$. (Note that $s_u = s_0(u)$.) Thus $s_u \geq 2s_v$. After that time u gets less than $(\alpha - 1)s_u$ new descendants. The ratio $s(u)/s(v)$ is minimized when all of these new descendants are also descendants of v. In this case the ratio equals $\frac{\alpha s_u}{s_v + (\alpha - 1)s_u} \geq \frac{\alpha}{\alpha - 1/2}$. Thus at any time $s(u) \geq (1 + \frac{1}{2\alpha - 1})s(v)$ as desired.

This relation gives the hypothesis (1) for fat preorder numbers. It is straightforward but tedious to verify that the left inequality of (3) implies the left inequality of (2). (Alternatively the reader can simply assume the specific values given, $\alpha = 5/4$, $e = 4$, $c = 6$, which satisfy (2) and (3) and suffice for the data structure.) Since all hypotheses for fat preorder numbers hold, the preceding discussion implies that the *ca* algorithm works correctly.

It is clear that most details of *add_leaf* work correctly. For instance note that the ancestor table for any node not in T_v remains valid after the recompression. The one fact that must be proved is that until a node u gets reorganized, its expansion interval is large enough to accommodate all new intervals. To do this fix a time. Consider a child v of u, and let s_v denote the current value of $s_0(v)$. (Note that $s(v)$ increases monotonically, although it can decrease to one when u gets reorganized.)

If v has been recompressed since u was reorganized, its previous value $s_0(v)$ was at most s_v/α. At any time the interval for v has size $cs_0(v)^e$. Thus the total size of all intervals used by v is strictly less than $cs_v^e(1 + (1/\alpha)^e + (1/\alpha)^{2e} + \dots) \leq \frac{cs(v)^e}{1-(1/\alpha)^e}$. The first paragraph of the proof indicates that $s(v) \leq (\alpha - 1/2)s_0(u)$. Now simple calculus shows that the total length of all intervals ever used by all children of u is less than $cs_0(u)^e \frac{(\alpha-1/2)^e+(1/2)^e}{1-(1/\alpha)^e}$. The left inequality of (3) guarantees that u's interval $[f(u)..g(u))$ is large enough to contain all intervals for all children of u, plus the number $f(u)$. (The latter follows from strictness of the above inequalities.) ∎

Now we extend these algorithms to allow *add_root* operations in addition to *add_leaf*. We show that the general incremental tree problem reduces to *add_leaf* and *ca* operations. We first extend the characteristic ancestor operation. For an arbitrary node r of T, let $ca(x, y, r)$ denote the characteristic ancestors of x and y when the root of T is changed to r; define $nca(x, y, r)$ similarly. All other terminology is unchanged, e.g., $ca(x, y)$ denotes the characteristic ancestors in T with its original root and "ancestor" refers to the original root. Observe that the path from x to r passes through $nca(x, r)$. This justifies the following algorithm for $ca(x, y, r)$ in terms of $ca(x, y)$: If $nca(x, r) = nca(y, r)$ then return $ca(x, y)$. Otherwise one of the two nodes $nca(x, r)$, $nca(y, r)$ equals $nca(x, y)$. Without loss of generality let $nca(y, r) = nca(x, y)$. Return $(b, b_x, p(b))$ where $ca(x, r) = (b, b_x, b_r)$.

It is now a simple matter to implement *add_root* operations in terms of *add_leaf*. The tree T maintains its current root as the value r. The operation *add_root*(y) is implemented as *add_leaf*(r, y) followed by $r \leftarrow y$. The algorithm for $ca(x, y)$ is $ca(x, y, r)$.

Lemma 3.5. The incremental-tree nearest common ancestors problem (with *ca* operations) can be solved in $O(m + n \log^2 n)$ time and $O(n \log n)$ space.

Proof. A *ca* operation uses $O(1)$ time. In *add_leaf*(x, y) examining each ancestor of y uses $O(\log n)$ time. Recompressing a node v uses $O(s(v))$ time for all processing except constructing the new ancestor tables, which uses $O(s(v) \log n)$ time. Hence the time for an *add_leaf* operation is $O(s(v) \log n)$ time. As noted in the preceding proof, when v is recompressed $(\alpha - 1)s_0(v)$ descendants of v have been added since the last reorganization of v. Charge the time for recompression to these new nodes, at the rate of $O(\log n)$ per node. The number of times a given node gets charged is at most its depth in D. This is at most $\log_\beta n$ by (1). Thus each node is charged $O(\log^2 n)$ time total, giving the time bound.

The only space that is nonlinear is for the ancestor tables. ∎

We close this section by noting that *add_root* can be implemented directly, without the general reduction. The main observation is that *add_root* changes the compressed tree in a simple way: Let T' be the result of performing *add_root*(y) on T, a tree with root x. If $|V(T)| > 1$ then y plus the heavy path with apex x in T forms a heavy path in T'. Thus $C(T')$ can be constructed from $C(T)$ by changing the name of the root from x to y and giving the root a new child named x. (This works when $|V(T)| = 1$ too.) This transformation is easily implemented in our data structure.

3.2. Multi-level incremental-tree algorithms.

This section improves the incremental-tree nearest common ancestors algorithm by using an algorithm with several levels. The improved algorithm runs in time $O(m + n)$.

We start with a framework for multi-level nearest common ancestor algorithms. (A similar framework is used in Section 3.3.) The terms *vertex* and *the incremental tree* T refer to the objects of the given problem, e.g., an operation *add_leaf*(x, y) makes vertex x the parent of vertex y in T. A multi-level algorithm works on a small number of levels designated $\ell = 1, \ldots, L$. (Later in this section we choose $L = 3$.) Each level ℓ has an associated partition of a subtree of T into subtrees called *ℓ-nodes*. Contracting all ℓ-nodes transforms the subtree of T into a tree T_ℓ. Thus there are $|V(T_\ell)|$ ℓ-nodes. Level ℓ has an associated partition of T_ℓ into subtrees called *ℓ-trees*. Each level has an algorithm and corresponding data structures to compute the characteristic ancestors of any two ℓ-nodes in the same ℓ-tree. (These characteristic ancestors are ℓ-nodes.)

We use this notation: If x is an ℓ-node, $\widehat{x}$ denotes the ℓ-tree containing x. Function p_ℓ denotes the parent function in T_ℓ.

An L-node is defined to be a vertex. Thus $T_L = T$. Each level ℓ has an integral size parameter μ_ℓ (dependant on n). Any ℓ-tree S has $|V(S)| \leq \mu_\ell$. Take $\mu_1 = n$ so there can be at most one 1-tree. Tree S is *full* if $|V(S)| = \mu_\ell$. A full ℓ-tree corresponds to an $(\ell - 1)$-node; more precisely the vertices contained in S form a subtree that is an $(\ell - 1)$-node. Any $(\ell - 1)$-node arises in this way. We usually say somewhat loosely that a full ℓ-tree "is" an $(\ell - 1)$-node. Any ℓ-tree with root node $x \neq r(T)$ has $p_\ell(x)$ in an $(\ell - 1)$-node.

The operation $ca(x, y)$ is performed using the recursive algorithm $c(x, y, \ell)$ which computes the characteristic ancestors of distinct ℓ-nodes x and y in T_ℓ. Thus for vertices x and y, $ca(x, y)$ equals $c(x, y, L)$. The outline of the algorithm is to first find the ℓ-tree S that contains $nca(x, y)$ and then use the level ℓ algorithm for characteristic ancestors in S.

We begin by considering the case where both ℓ-trees $\widehat{x}$ and $\widehat{y}$ are $(\ell - 1)$-nodes. To do the first step, if $\widehat{x} = \widehat{y}$ then $\widehat{x}$ is the desired ℓ-tree; return $ca_{\widehat{x}}(x, y)$. Otherwise $\widehat{x}$ and $\widehat{y}$ correspond to two

$(\ell-1)$-nodes. Set $(a, a_x, a_y) \leftarrow c(\widehat{x}, \widehat{y}, \ell-1)$. Thus the ℓ-tree a contains $nca(x,y)$. For $z = x, y$ set b_z to the first ℓ-node ancestor of z in a: If $a_z = a$ then $b_z = z$ else $b_z = p_\ell(r(a_z))$ ($r(a_z)$ is the root of the ℓ-tree a_z). If $b_x \neq b_y$ then the desired characteristics ancestors are $ca_a(b_x, b_y)$. Compute this using the level ℓ *ca* algorithm for ℓ-tree a. If $b_x = b_y$ then $b_x = nca(x,y)$. It is easy to find the two other characteristic ancestors – they are among the nodes b_x, $r(a_x)$ and $r(a_y)$.

It remains to consider the case when $\widehat{x}$ or $\widehat{y}$ is not an $(\ell-1)$-node. As above, if $\widehat{x} = \widehat{y}$ then return $ca_{\widehat{x}}(x,y)$. Otherwise for $z = x, y$ set c_z to the first ℓ-node ancestor of z in an $(\ell-1)$-node: If z is in such a node then $c_z = z$ else $c_z = p_\ell(r(\widehat{z}))$. If $c_x \neq c_y$ then the desired characteristic ancestors are $ca(c_x, c_y)$, which can be found by the procedure for the first case. If $c_x = c_y$ then $c_x = nca(x,y)$ and the other characteristic ancestors are among the nodes c_x, $r(\widehat{x})$ and $r(\widehat{y})$. This concludes the algorithm for *ca*. Correctness is clear from the accompanying discussion.

Next consider *add_leaf*(x,y). We use a similar recursive algorithm $al(x,y,\ell)$ where ℓ-node $x \in T_\ell$ is to be made the parent of a new ℓ-node y. If $\widehat{x}$ is full then make y a singleton ℓ-tree, make x the parent of node y and halt. Otherwise $\widehat{x}$ is not full; execute the level ℓ algorithm *add_leaf*(x,y). If $\widehat{x}$ is still not full then halt. Otherwise if $\widehat{x}$ is the unique ℓ-tree then make $\widehat{x}$ the unique $(\ell-1)$-node and a singleton $(\ell-1)$-tree, and halt. In the remaining case observe that $p_\ell(r(\widehat{x}))$ is in an $(\ell-1)$-node, say w. Call $al(w, \widehat{x}, \ell-1)$ to complete the processing. This algorithm is correct because it clearly preserves the defining properties of the data structure.

For incremental trees take $L = 3$ levels. Set $\mu_3 = \lceil (\log^{(2)} n)^2 \rceil$ and $\mu_2 = \lceil \log^2 n \rceil$. Levels two and one use the incremental-tree algorithm of Section 3.1. The algorithm for level three is specified below. First however observe that levels two and one both use $O(m+n)$ time and $O(n)$ space. To prove this, Lemma 3.5 shows that an ℓ-tree with k ℓ-nodes processes p *ca* operations in $O(p + k \log^2 \mu_\ell)$ time. Any level $\ell < L$ has at most $\frac{n}{\mu_{\ell+1}}$ ℓ-nodes. Thus the total time on level ℓ is $O(m + \frac{n \log^2 \mu_\ell}{\mu_{\ell+1}})$. For $\ell = 1, 2$ we have $\frac{\log^2 \mu_\ell}{\mu_{\ell+1}} = O(1)$. Thus levels one and two use $O(m+n)$ time. A similar argument using Lemma 3.5 shows that the space for level $\ell = 1, 2$ is $O(\frac{n \log \mu_\ell}{\mu_{\ell+1}}) = O(n)$.

It remains only to specify the algorithm for level 3. We use a technique similar to microsets [GT85]. Each node of the tree has an identifier – an integer between one and μ_3. The ith node added to the tree is assigned the identifier i. Each node x has an *ancestor list*, denoted *ancestor*$[x]$. This is the sequence of identifiers of its ancestors, starting with the root and ending with x. Since a node has at most μ_3 ancestors, an ancestor list can be clearly be represented by $O((\log^{(2)} n)^3)$ bits. Since we assume a random access machine with a word size of $\log n$ bits, *ancestor*$[x]$ fits in one word. Store each ancestor list left-justified in its word, with each identifier written as a string of precisely $\lfloor \log \mu_3 \rfloor + 1$ bits.

The algorithm for *add_leaf*(x, y) constructs *ancestor*$[y]$ by adding y's identifier in the appropriate position to *ancestor*$[x]$. This can be done in $O(1)$ time using arithmetic operations. The algorithm for $ca(x, y)$ forms the boolean *exclusive or* of *ancestor*$[x]$ and *ancestor*$[y]$. The most significant bit of the result occurs within the bit field that stores a_x in *ancestor*$[x]$ and a_y in *ancestor*$[y]$. All boolean operations needed – *exclusive or*, finding the most significant bit, and recovering the appropriate fields for a_x and a_y, can be done in $O(1)$ time by table look-up. The appropriate tables are generated in $O(n)$ time. A more detailed discussion of similar algorithms involving table look-up can be found in [AHU, GT85].

This discussion implies that level three uses $O(m + n)$ time and $O(n)$ space. This completes the three level algorithm.

Theorem 3.1. The incremental-tree nearest common ancestors problem (with *ca* operations) can be solved in $O(m + n)$ time and $O(n)$ space. ∎

We close this section by sketching a simpler version of our algorithm. It achieves the same asymptotic efficiency but applies only to the problem of nearest common ancestors for static trees. In this case the data structure can be simplified from three levels to two. This algorithm seems to be simpler than the static tree algorithm of [HT], which has the same asymptotic efficiency but uses three levels (called "plies").

Take $\mu_2 = \lceil \frac{\log n}{2} \rceil$. Construct the 1- and 2-trees recursively as follows: If the tree S has less than μ_2 nodes then make it a 2-tree. Otherwise let S_0 be a subtree containing $r(S)$ and having μ_2 nodes; make S_0 a 2-tree and a 1-node, and process the trees of forest $S - S_0$ recursively.

The unique tree on level one has $O(n/\log n)$ nodes. Use the incremental-tree algorithm of Section 3.1 on it. Lemma 3.3 shows the preprocessing time and space on level one is $O(n)$.

Level two uses a different microset data structure, related to the Euler tour technique of [TV]. Represent a tree S of b nodes by a string β of balanced parentheses of length $2b - 2$. β represents a depth-first traversal of S – "(" corresponds to when the search descends along an edge for the first time, ")" corresponds to when the search ascends along an edge, having explored a subtree. Each node x of S has a canonical representation as the string β_x, the shortest prefix of β that leads to it. Observe that $nca(x, y)$ can be found as follows: Without loss of generality assume that $x \neq y$ and β_y is longer than β_x. Let γ be the suffix of β_y following β_x. Let δ be the shortest prefix of γ ending with a "(" that does not have a matching ")" in the entire string γ. (γ may contain some unmatched right parentheses. However δ still well-defined.) It is easy to see that δ exists, and $nca(x, y)$ corresponds to the string $\beta_x \delta'$, where δ' is δ with its ending left parenthesis removed.

For a tree S with at most μ_2 nodes, β fits into one word. It is not hard to design a set of tables that can be precomputed in $O(n)$ time, such that for any tree S, $ca_S(x,y)$ can be found in $O(1)$ time from β_x and β_y. This gives a two level, static tree algorithm that uses $O(n)$ preprocessing time, $O(n)$ space, and performs a *ca* query in $O(1)$ time.

3.3. General link operations.

This section gives a multi-level algorithm for the nearest common ancestors with linking problem. The algorithm processes m *nca* and *link* operations on a set of n nodes in time $O(m\alpha(m,n)+n)$.

Define Ackermann's function $A(i,j)$ for $i,j \geq 1$ by

$$A(1,j) = 2^j, \text{ for } j \geq 1;$$
$$A(i,1) = 2, \text{ for } i \geq 2;$$
$$A(i,j) = A(i-1, A(i,j-1)), \text{ for } i,j \geq 2.$$

Define two inverse functions,

$$a(i,n) = \min\{j \mid A(i,j) \geq n\};$$
$$\alpha(m,n) = \min\{i \mid A(i, 4\lceil m/n \rceil) \geq n\}, \text{ for } m,n \geq 1.$$

These definitions differ slightly from those of [T83] but this does not change asymptotic estimates. The most significant difference is that our function $A(i,1)$ is constant compared to a rapidly growing function in [T83]. This makes for a more convenient treatment of the base case in our algorithms. We use some simple properties of Ackermann's function including these inequalities:

$$\begin{aligned} &A(i,j+1) \geq 2A(i,j), \text{ for } i,j \geq 1; \\ &A(i+1,j) \geq A(i,2j), \text{ for } i \geq 1, j \geq 4. \end{aligned} \tag{4}$$

We use the incremental tree data structure of Section 3.2. Call a tree that is represented by this data structure an *incremental tree.*

The approach is similar to that of [G85b] for a list splitting problem. We construct a multi-level algorithm recursively. For $\ell \geq 1$ the algorithm with ℓ levels is denoted $\mathcal{A}_\ell$. It calls $\mathcal{A}_{\ell-1}$ if $\ell > 1$. Algorithm $\mathcal{A}_\ell$ runs in time $O(m\ell + na(\ell,n))$. Assume that any value $A(i,j)$ that is at most n can be found in $O(1)$ time. This assumption is justified later.

Algorithm $\mathcal{A}_\ell$ is said to work on level ℓ. The terms *node* and *link tree* refer to the objects manipulated by $\mathcal{A}_\ell$, i.e., an instruction *link*(x,y) operates on nodes x,y to produce a new link tree.

Level ℓ has the following structure. There are $a(\ell,n)$ *universes* u, $u = 0,\ldots,a(\ell,n)-1$. Each link tree T is in some universe u. If $|V(T)| < 4$ then $u = 0$.

If $|V(T)| \geq 4$ then T is in a universe $u \geq 1$, chosen so that $|V(T)| \in [2A(\ell,u)..2A(\ell,u+1))$. (This is possible since $A(\ell,1) = 2$.) An *ℓ-tree (in universe u)* is a subtree that has at least $2A(\ell,u)$ nodes. It is represented as an incremental tree. The nodes of T are partitioned into ℓ-trees. If T contains more than one ℓ-tree then T, with each ℓ-tree contracted, is represented using the data structure for algorithm $\mathcal{A}_{\ell-1}$. If $\ell = 1$ observe that T has only one ℓ-tree, since T has less than $2A(1,u+1) = 2^{u+2}$ nodes and an ℓ-tree has at least $2A(1,u) = 2^{u+1}$ nodes.

Algorithm $\mathcal{A}_\ell$ uses the following data structure for level ℓ. Each link tree and ℓ-tree is represented by its root. A link tree T is stored using parent pointers and children lists. If r is the root of T then $s(r)$ equals the size of T. For any node x, $u(x)$ equals the universe that contains x's link tree; if $u(x) > 0$ then $\widehat{x}$ designates the ℓ-tree containing x.

We implement the operation $link(x,y)$ by a recursive algorithm $l(r,x,y)$, where r is the root of the link tree containing x. (Recall that y is the root of its link tree.) To make the initial call to l we must find the root r. This is done by a straightforward recursive algorithm: Suppose $u(x) > 0$. First set R to the ℓ-tree containing the root. To do this set $R = \widehat{x}$; if this is not the desired ℓ-tree then recursively compute R as the root of the tree containing $\widehat{x}$ on level $\ell - 1$. Then return the root of the ℓ-tree R.

The algorithm for $l(r,x,y)$ is as follows. Let T_r and T_y denote the link trees with root r and y respectively, before the link operation. Set the parent of y to x and add y to the child list of x. Increase $s(r)$ by $s(y)$. Let $u = \max\{u(x), u(y)\}$. Execute the first one of the following cases that applies and then return.

Case $s(r) \geq 2A(\ell,u+1)$: Make the new link tree T into an ℓ-tree in universe $u+1$: Traverse T top-down; when visiting a node v do an *add_leaf* operation to add v to the new incremental tree. Discard the data structures for T_r and T_y.
Case $u(x) > u(y)$: Traverse T_y top-down, doing *add_leaf* operations to add each node to the incremental tree $\widehat{x}$. Discard the data structure for T_y.
Case $u(x) < u(y)$: Traverse the path from x to r, doing *add_root* operations to add each node to the incremental tree $\widehat{y}$. Then traverse T_r top-down, doing *add_leaf* operations to add the other nodes to $\widehat{y}$. Discard the data structure for T_r and set $u(r)$ to u.
Case $u(x) = u(y)$: If $u > 0$ then do $l(\widehat{r},\widehat{x},\widehat{y})$ in the data structure for $\mathcal{A}_{\ell-1}$. ∎

This algorithm is correct because it preserves the invariants of the data structure. (We assume the bookkeeping fields $u(v)$ and $\widehat{v}$ are updated when node v is added to a new incremental tree.) In particular note these points. In the first case the new link tree belongs in universe $u+1$ because

$s(r) < 4A(\ell, u+1) \leq 2A(\ell, u+2)$, by (4). In the second case the incremental tree $\widehat{x}$ exists, since x is in a positive universe. Similarly in the third case $\widehat{y}$ exists. The fourth case always has $\ell > 1$. (This guarantees that the algorithm $\mathcal{A}_{\ell-1}$ that is called actually exists.) This is because if $\ell = 1$ and $u(r) = u(y)$ then the first case applies, since $s(r) \geq 2^{u+2} = 2A(\ell, u+1)$.

The algorithm for $ca(x, y)$ is trivial in universe zero. In positive universes it is the multi-level algorithm $c(x, y, \ell)$ given in Section 3.2. Each ℓ-tree is, in the terminology of that section, an $(\ell - 1)$-node. Hence the first case of the c algorithm is used. It executes $c(\widehat{x}, \widehat{y}, \ell - 1)$ and then finds the desired characteristic ancestors by executing the incremental tree ca algorithm on the appropriate ℓ-tree. The details are in Section 3.2.

Lemma 3.6. Algorithm $\mathcal{A}_\ell$ executes a sequence of m *ca* and *link* operations on a set of n nodes in $O(m\ell + na(\ell, n))$ time and $O(n)$ space.

Proof. First consider the time. A *ca* query uses $O(\ell)$ time in a positive universe, since $O(1)$ time is spent on each of ℓ levels of recursion. The time is $O(1)$ in universe zero.

Estimate the time for *links* as follows. Charge each *link* operation $O(\ell)$ time to account for the initial computation of root r and the ℓ levels of recursion and associated processing in routine l. Now observe that the rest of the time for l is proportional to the number of *add_leaf* and *add_root* operations (including those in recursive calls). This relies on Theorem 3.1, which shows that each such operation uses (amortized) time $O(1)$. Thus it suffices to show that the total number of *add_leaf* and *add_root* operations is $O(na(\ell, n))$. In fact we show by induction on ℓ that there are at most $2na(\ell, n)$ such operations.

Consider first all *add_leaf* and *add_root* operations except those done in recursive calls. Observe that each such operation is done for a node previously in a lower universe. Thus at most one operation is done for each node in each universe. This gives at most $na(\ell, n)$ operations total. (In particular this establishes the base case of the induction, $\ell = 1$.)

To bound the operations in recursive calls to $\mathcal{A}_{\ell-1}$, consider any universe $u > 0$. By induction the number of operations associated with each ℓ-tree in universe u is at most twice the quantity

$$a(\ell - 1, 2A(\ell, u+1)/2A(\ell, u)) \leq a(\ell - 1, A(\ell, u+1)) = a(\ell - 1, A(\ell - 1, A(\ell, u))) = A(\ell, u).$$

There are at most $n/2A(\ell, u)$ ℓ-trees in universe u. Thus the total number of operations in recursive calls associated with universe u is at most n. This gives at most $na(\ell, n)$ operations total.

Adding together the two cases shows there are at most $2na(\ell, n)$ *add_leaf* and *add_root* operations as desired.

Next consider the space. At any time, the space is proportional to the number of nodes on all levels. We show by induction on ℓ that this number is at most $2n$. There are n nodes on level ℓ. This establishes the base case $\ell = 1$, since there are no lower levels. Suppose $\ell > 1$. For $u \geq 1$, over the entire algorithm there are at most $n/2A(\ell, u)$ distinct ℓ-trees in universe u. By induction the total number of nodes associated with universe u, on levels $\ell - 1$ and lower, is at most $n/A(\ell, u)$. Since $A(\ell, u) \geq 2^u$, the total number of nodes is at most $n + (n/2 + n/4 + \dots) \leq 2n$. This proves the space bound. ∎

The remaining issue is how to choose the number of levels ℓ. Suppose first that m and n are known when the algorithm begins. Take $\ell = \alpha(m, n)$. Observe that $a(\alpha(m, n), n)) \leq 4\lceil m/n \rceil$ since $A(\alpha(m, n), 4\lceil m/n \rceil) \geq n$. Thus the total time for algorithm $\mathcal{A}_\ell$ is $O(m\alpha(m, n) + n)$, the desired bound.

We must justify the assumption that any value $A(i, j)$ that is at most n can be found in $O(1)$ time. As part of the initialization the algorithm computes a table $ackermann[i, j]$ for $i, j \leq \log n$; if $A(i, j) \leq n$ then $ackermann[i, j] = A(i, j)$, else $ackermann[i, j] = \emptyset$. Thus all desired values of Ackermann's function can be found by table look-up. The desired value of $\ell = \alpha(m, n)$ can also be found. The time to initialize the table is and find ℓ is clearly $O(\log^2 n)$.

Now we show that the same time bound can be achieved when m and n are not known in advance. In this setting we allow the operation $make\_node(x)$ which creates a new node x in a singleton tree. At any time n denotes the total number of *make_node* operations and m denotes the total number of *ca* and *link* operations.

The procedure works by using algorithm $\mathcal{A}_\ell$ where ℓ is repeatedly modified. The sequence of operations is divided into *periods*. The value of n and m at the beginning of the ith period is denoted n_i and m_i respectively. Initialize $i = n_0 = m_0 = 0$ and $\ell = 1$.

After processing an operation, if $m \geq 1$ and either $n \geq 2n_i$ or $m \geq \max\{2m_i, n\}$ then end the current period i as follows. Set $i \leftarrow i + 1$, $n_i \leftarrow n$, $m_i \leftarrow m$. If $\ell \neq \alpha(m, n)$ then assign $\ell \leftarrow \alpha(m, n)$ and reorganize the entire data structure to use algorithm $\mathcal{A}_\ell$. Do this by making each link tree T an incremental tree and placing it in the correct universe.

Note that the reorganization procedure is correct – T does not have a data structure on level $\ell - 1$. Thus the time to reorganize n nodes is $O(n)$. (This includes the time to compute a new table $ackermann[i, j]$, to find ℓ and find the universe for each incremental tree. The table stores all values $A(i, j) \leq 2n$.) The time to end a period that does not reorganize the data structure is $O(1)$.

Now we analyze the total time for the algorithm. Period zero ends after the first *ca* or *link* operation. Clearly it uses $O(n)$ time, so we restrict attention to periods $i \geq 1$. A reorganization decreases ℓ by at most one. To show this consider a period $i \geq 1$ with parameters n_i, m_i and $\ell = \alpha(m_i, n_i)$. If ℓ decreases then $\lceil \frac{m_{i+1}}{n_{i+1}} \rceil > \lceil \frac{m_i}{n_i} \rceil$. Thus $m_{i+1} > n_{i+1}$ and inspecting the algorithm shows $m_{i+1} \leq 2m_i$. Using inequalities (4) shows that

$$A(\ell - 2, 4\left\lceil \frac{m_{i+1}}{n_{i+1}} \right\rceil) \leq A(\ell - 2, 4\left\lceil \frac{2m_i}{n_i} \right\rceil) \leq A(\ell - 2, 8\left\lceil \frac{m_i}{n_i} \right\rceil) \leq A(\ell - 1, 4\left\lceil \frac{m_i}{n_i} \right\rceil) < n_i.$$

Thus ℓ decreases by at most one. A similar calculation proves that ℓ increases by at most two, but we do not need this fact.

Theorem 3.2. A sequence of m *nca* and *link* operations on a universe of n nodes can be processed in time $O(m\alpha(m, n) + n)$ and space $O(n)$.

Proof. We need only prove the time bound when m and n are not known in advance. For convenience assume that the last operation of the algorithm is followed by a reorganization. This can be ensured by extending the input sequence with a sufficient number of *make_node* operations; this at most doubles n and so does not change the asymptotic time bound.

We account for the time by charging each operation a time-varying amount. Define a charge of one unit to be large enough to account for any constant amount of computing. The following invariants are maintained. Each *make_node* operation is charged at most one unit if it is in the current period else at most two units. Each *link* or *ca* operation is charged at most ℓ units if it is in the current period else at most $\ell + 3$ units, where ℓ denotes the current value of that parameter. After each reorganization the total time (for the entire algorithm) is at most the number of units charged. Clearly these invariants imply the total time is $O(m\ell + n)$. Since the last reorganization ensures $\ell = \alpha(m, n)$ this equals the desired bound.

Charge the operations as follows. When a *make_node* operation is executed charge it one unit; this accounts for the $O(1)$ time it uses. When a *ca* or *link* operation is executed charge it ℓ units. This accounts for all the time used by *cas* and part of the time used by *links*. The remaining time for *links* is associated with *add_leaf* and *add_root* operations, as indicated in the above timing analysis. It is charged in the next reorganization. Consider the end of period i. If there is no reorganization then no new charges are made. If the data structure is reorganized then we make charges for $O(m_i + n_i)$ units of computing.

Let us show that in accordance with the invariants, after a reorganization the total time has been accounted for. The new charges must account for the remaining time for *links* and also the

time for reorganization. The former is certainly bounded by $O(n_{i+1}a(\ell, n_{i+1}))$ where $\ell = \alpha(m_i, n_i)$. This expression also bounds the time for reorganization, $O(n_{i+1})$. To show this expression is the desired quantity $O(m_i + n_i)$ it suffices to show that $a(\ell, n_{i+1}) \leq 4\lceil \frac{m_i}{n_i} \rceil + 1$, since $n_{i+1} \leq 2n_i$. The latter follows since $A(\ell, 4\lceil \frac{m_i}{n_i} \rceil + 1) \geq 2A(\ell, 4\lceil \frac{m_i}{n_i} \rceil) \geq 2n_i \geq n_{i+1}$.

To make the charges when the data structure is reorganized, consider two cases. Suppose first that $n_{i+1} = 2n_i$. Observe that the value of ℓ increases in the reorganization. For this it suffices to show $\lceil \frac{m_{i+1}}{n_{i+1}} \rceil \leq \lceil \frac{m_i}{n_i} \rceil$, since this inequality implies that ℓ does not decrease. If $m_{i+1} \leq n_{i+1}$ then $\lceil \frac{m_{i+1}}{n_{i+1}} \rceil = 1$, which gives the inequality. The other possibility is that $m_{i+1} \leq 2m_i$. This implies $\frac{m_{i+1}}{n_{i+1}} \leq \frac{m_i}{n_i}$ so again the inquality holds.

Make the timing charge $O(m_i + n_i)$ as follows. For the charge of $O(n_i)$, charge one unit to each of the n_i new *make_node* operations in period i. Each such operation is now charged two, preserving the invariant. For the charge of $O(m_i)$, charge one more unit to each *link* or *ca* operation in all periods up to i. This preserves the invariant since ℓ increases.

Now suppose $m_{i+1} \geq \max\{2m_i, n_{i+1}\}$. The number of new operations, $m_{i+1} - m_i$, satisfies $m_{i+1} - m_i \geq m_i$ and $m_{i+1} - m_i \geq m_{i+1}/2 \geq n_{i+1}/2 \geq n_i/2$. Thus the timing charge $O(m_i + n_i)$ can be made by charging one more unit to each *ca* or *link* operation in period i.

In addition we must preserve the invariant for *link* and *ca* operations. Suppose the value of ℓ decreases in the reorganization. As indicated above, it decreases to $\ell - 1$. Charge one less unit to each of the m_i *link* or *ca* operations before period i, and one more unit to each such operation in period i. This does not decrease the total charge since as already noted $m_{i+1} - m_i \geq m_i$. It preserves the invariant, since the operations before period i are now charged at most $\ell + 2$, the operations in period i are charged exactly $\ell + 2$, and $\ell + 2 = (\ell - 1) + 3$. If the value of ℓ increases in the reorganization, the invariant holds *a fortiori*. ∎

The multi-level method used for Theorem 3.2 can be applied to achieve the same time and space bounds for several other problems. As mentioned above, [G85b] applies it to solve the list splitting problem that arises in *expand* steps of Edmonds' algorithm. The technique was recently rediscovered by Han La Poutré: [LaP] presents a multi-level algorithm for the set merging problem (this application is noted in [G85b, p. 99]) and a result similar to Theorem 3.2 was independently arrived at [J.A. La Poutré, personal communication]. Other applications include the static cocycle problem introduced in [GS], both for graphic matroids and the job scheduling matroid; the former is useful for various problems involving spanning trees. We will discuss these and other applications in a forthcoming paper.

Acknowledgments.

Thanks to Bob Tarjan for some very fruitful early conversations, and also to Jim Driscoll.

References.

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

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

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

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

[FS] M.L. Fredman and M.E. Saks, "The cell probe complexity of dynamic data structures", *Proc. 21st Annual ACM Symp. on Theory of Comp.,* 1989, pp. 345–354.

[FT] M.L. Fredman and R.E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", *J. ACM, 34,* 3, 1987, pp. 596–615.

[G73] H.N. Gabow, "Implementations of algorithms for maximum matching on nonbipartite graphs", Ph. D. Dissertation, Comp. Sci. Dept., Stanford Univ., Stanford, Calif., 1973.

[G83] H.N. Gabow, "An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems", *Proc. 15th Annual ACM Symp. on Theory of Comp.,* 1983, pp 448-456.

[G85a] H.N. Gabow, "Scaling algorithms for network problems", *J. Comp. and System Sci., 31,* 2, 1985, pp. 148–168.

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

[GGS] H.N. Gabow, Z. Galil and T.H. Spencer, "Efficient implementation of graph algorithms using contraction", *J. ACM, 36,* 3, 1989, pp. 540–572.

[GGST] H.N. Gabow, Z. Galil, T.H. Spencer and R.E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs", *Combinatorica 6,* 2, 1986, pp. 109–122.

[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. Comput., 15,* 1, 1986, pp. 120–130.

[GS] H.N. Gabow and M. Stallmann, "Efficient algorithms for graphic matroid intersection and parity", *Automata, Languages and Programming: 12th Colloquium, Lecture Notes in Computer Science 194*, W. Brauer, ed., Springer-Verlag, 1985, pp. 210–220.

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

[GT89] H.N. Gabow and R.E. Tarjan, "Faster scaling algorithms for general graph matching problems", *J. ACM,* to appear.

[HT] D. Harel and R.E. Tarjan, "Fast algorithms for finding nearest common ancestors", *SIAM J. Comput., 13,* 2, 1984, pp. 338–355.

[K55] H.W. Kuhn, "The Hungarian method for the assignment problem", *Naval Res. Logist. Quart., 2,* 1955, pp. 83–97.

[K56] H.W. Kuhn, "Variants of the Hungarian method for assignment problems", *Naval Res. Logist. Quart., 3,* 1956, pp. 253–258.

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

[LaP] J.A. La Poutré, "New techniques for the union-find problem", *Proc. First Annual ACM-SIAM Symp. on Disc. Algorithms,* 1990, pp. 54–63.

[NW] G.L. Nemhauser and G.M. Weber, "Optimal set partitioning, matchings and Lagrangian duality", *Naval Res. Logist. Quart., 26,* 1979, pp. 553–563.

[LP] L. Lovász and M.D. Plummer, *Matching Theory*, North-Holland Mathematic Studies 121, North-Holland, New York, 1986.

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

[SlT] D.D. Sleator and R.E. Tarjan, "A data structure for dynamic trees", *J. Comp. and System Sci., 26,* 1983, pp. 362–391.

[SV] B. Schieber and U. Vishkin, "On finding lowest common ancestors: simplification and parallelization", *SIAM J. Comput., 17,* 6, 1988, pp. 1253–1262.

[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, Philadelphia, PA., 1983.

[TV] R.E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm", *SIAM J. Comput., 14,* 4, 1985, pp. 862–874.

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

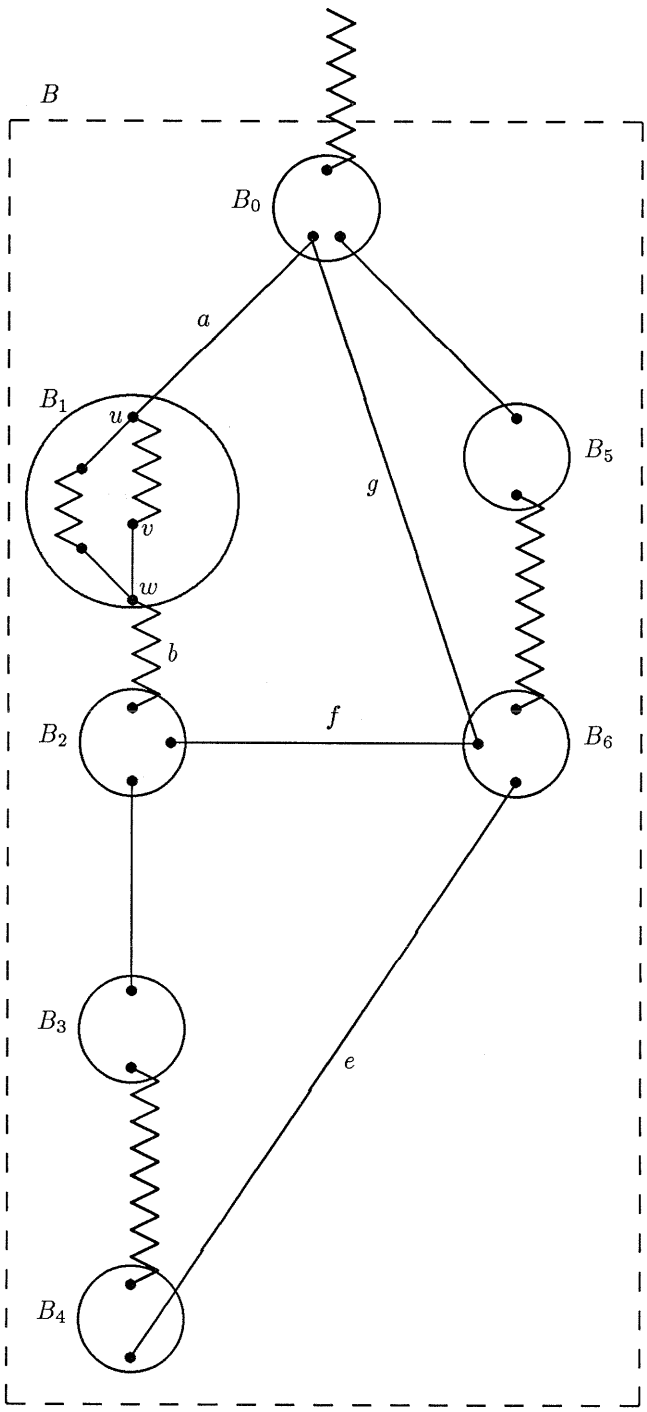


Figure 1.
Blossoms in a search of Edmonds' algorithm.