An $O(n \log n)$ Algorithm for the Maximum Agreement Subtree Problem for Binary Trees*

Richard Cole† Martin Farach‡ Ramesh Hariharan§

Teresa Przytycka¶ Mikkel Thorup ‖

Abstract

The Maximum Agreement Subtree problem is the following: given two trees whose leaves are drawn from the same set of items (e.g., species), find the largest subset of these items so that the portions of the two trees restricted to these items are isomorphic. We consider the case which occurs frequently in practice, i.e., the case when the trees are binary, and give an $O(n \log n)$ time algorithm for this problem.

1 Introduction

Suppose we are given two rooted trees T_1 and T_2 with n leaves each. The internal nodes of each tree have at least two children each. The leaves in each tree are labelled with the same set of labels and further, no label occurs more than once in a particular tree. An *agreement subtree* of T_1 and T_2 is defined as follows. Let L_1 be a subset of the leaves of T_1 and let L_2 be the subset of those leaves of T_2 which have the same labels as leaves in L_1. The subtree of T_1 *induced* by L_1 is an agreement subtree of T_1 and T_2 if and only if it is *isomorphic* to the subtree of T_2 induced by L_2. The Maximum Agreement Subtree problem (henceforth called *MAST*) asks for the largest agreement subtree of T_1 and T_2.

We need to define the terms *induced subtree* and *isomorphism* used above. Intuitively, the subtree of T_1 induced by L_1 is the subtree of T_1 restricted to leaves in L_1, with branching information relevant to L_1 preserved. More formally, it is the tree obtained by the following procedure. Repeatedly remove from T_1 all leaves not in L_1; remove also the edges incident upon these leaves. Next, replace

*Work supported by NSF grants CCR-9202900 and CCR-9503309.

†Courant Institute of Mathematical Sciences, NYU.

‡DIMACS.

§Indian Institute of Science, Bangalore. Work done when the author was at Max Planck Institut für Informatik, Germany, and when visiting NYU.

¶University of Maryland, College Park.

‖University of Copenhagen, Denmark.

all maximal paths of degree 2 nodes with single edges; each internal node in the resulting tree will have at least 2 children. The resulting tree is the induced subtree.

Intuitively, two trees are isomorphic if the children of each node in one of the trees can be reordered so that the leaf labels in each tree occur in the same order and the shapes of the two trees become identical. Formally, trees F and G are said to be isomorphic if the following conditions hold. If F and G are singleton trees, i.e., leaves, then they are isomorphic if they have the same labels. Otherwise, F and G are isomorphic if their roots have the same number of children and, further, there exists a one-to-one mapping $f()$ from the children of the root of F to those of the root of G so that for each child x of the root of F, the subtree of F rooted at x is isomorphic to the subtree of G rooted at $f(x)$.

Motivation. The *MAST* problem arises naturally in biology and linguistics as a measure of consistency between two evolutionary trees over species and languages, respectively. An evolutionary tree for a set of *taxa*, either species or languages, is a rooted tree whose leaves represent the taxa and whose internal nodes represent ancestor information. It is often difficult to determine the true phylogeny for a set of taxa, and one way to gain confidence in a particular tree is to have different lines of evidence supporting that tree. In the biological taxa case, one may construct trees from different parts of the DNA of the species. These are known as *gene trees*. For many reasons, these trees need not entirely agree, and so one is left with the task of finding a consensus of the various gene trees. The maximum agreement subtree is one method of arriving at such a consensus. Notice that all gene trees must be binary, since DNA always replicates by a binary branching process. Therefore, the case of binary trees is of great interest.

Another application arises in automated translation between two languages [GY95]. The two trees are the parse trees for the same meaning sentences in the two languages. A complication that arises in this application (due in part to imperfect dictionaries) is that words need not be uniquely matched, i.e., a word at the leaf of one tree could match a number (usually small) of words at the leaves of the other tree. The aim is to find a maximum agreement subtree; this is done with the goal of improving context-using dictionaries for automated translation. So long as each word in one tree has only a constant number of matches in the other tree (possibly with differing weights), the algorithm given here can be used and its performance remains $O(n \log n)$. More generally, if there are m word matches in all, the performance becomes $O((m+n)\log n)$. Note however, that if there are two collections of equal meaning words in the two trees of sizes k_1 and k_2 respectively, they induce $k_1 k_2$ matches.

Previous Work. Finden and Gordon [FG85] gave a heuristic algorithm for the *MAST* problem on binary trees which had an $O(n^5)$ running time and did not guarantee an optimal solution. Kubicka, Kubicki and McMorris [KKM92] gave an $O(n^{(.5+\epsilon)\log n})$ algorithm for the same problem. The first polynomial time algorithm for this problem was given by Steel and Warnow [SW93]; it had a running time of $O(n^2)$. Steel and Warnow also considered the case of non-binary and unrooted trees. Their algorithm takes $O(n^2)$ time for fixed degree rooted and unrooted trees and $O(n^{4.5} \log n)$ for arbitrary degree rooted and unrooted trees. They also give a linear reduction from the rooted to the unrooted case. Farach and Thorup gave an $O(nc^{\sqrt{\log n}})$ time algorithm for the *MAST* problem on binary trees; here c is a constant greater than 1. For arbitrary degree trees, their algorithm takes $O(n^2 c^{\sqrt{\log n}})$ time for the unrooted case [FT95] and $O(n^{1.5} \log n)$ time for the rooted case [FT94]. Farach, Przytycka, and Thorup [FPT95a] obtained an $O(n \log^3 n)$ algorithm for the *MAST* problem on binary trees. Kao [Ka95] obtained an algorithm for the same problem which takes $O(n \log^2 n)$ time. This algorithm takes $O(\min\{nd^2 \log d \log^2 n, nd^{\frac{3}{2}} \log^3 n\})$ for degree d trees. Finally, Cole and Hariharan [CR96] improved the algorithm from [FPT95a] to an $O(n \log n)$ algorithm.

The *MAST* problem for more than two trees has also been studied. Amir and Keselman [AK94] showed that the problem is NP-hard for even 3 unbounded degree trees. However, polynomial bounds are known [AK94, FPT95b] for three or more bounded degree trees.

Our Contribution. This paper is the combined journal version of [FPT95b] and [CR96] and presents an $O(n \log n)$ algorithm for the *MAST* problem for two binary trees.

The $O(n \log^3 n)$ algorithm of [FPT95a] can be viewed as taking the following approach (although the authors do not describe it this way). It identifies two special cases and then solves the general case by interpolating between these cases.

Special Case 1: The internal nodes in both trees form a path. The *MAST* problem reduces to essentially a size n Longest Increasing Subsequence Problem in this case. As is well known, this can be solved in $O(n \log n)$ time.

Special Case 2. Both trees T_1 and T_2 are complete binary trees. Consider a node v in T_2. Let L_2 be the set of leaves which are descendants of v. Let L_1 be the set of leaves in T_1 which have the same labels as the leaves in L_2. Only certain nodes u in T_1 can be usefully mapped to v, in the sense that the subtree of T_1 rooted at u and the subtree of T_2 rooted at v have a non-empty Agreement Subtree. These nodes u are the ancestors of the leaves in L_1. There are $O(n \log^2 n)$ such pairs (u, v). For each pair, computing the *MAST* takes $O(1)$ time, as it is simply a question of deciding the best way of pairing their children.

The interpolation process takes a centroid decomposition of the two trees and compares pairs of centroid paths, rather than individual nodes as in the complete tree case. The comparison of a pair of centroid paths requires finding matchings with special properties in appropriately defined bipartite graphs, a non-trivial generalization of the Longest Increasing Subsequence problem. This process creates $O(n \log^2 n)$ interesting (u, v) pairs, each of which takes $O(\log n)$ time to process.

In [CR96] two improvements are given, each of which gains a $\log n$ factor.

Improvement 1. The complete tree special case is improved to $O(n \log n)$ time as follows. The same notation as above is used: v, L_1, L_2. In fact, the only nodes that need be compared to v are those nodes u in the subtree of T_1 induced by L_1. This reduces the number of interesting pairs (u, v) to $O(n \log n)$. Again, processing a pair takes $O(1)$ time (this is less obvious, for identifying the descendants of u which root the subtrees with which the two subtrees of v can be matched is non-trivial). Constructing the above induced subtree itself can be done in $O(|L_1|)$ time, as will be detailed later. The basic tool here is to preprocess trees T_1 and T_2 in $O(n)$ time so that least common ancestor queries can be answered in $O(1)$ time.

Improvement 2: As in [FPT95a], when the trees are not complete binary trees, we take centroid paths and match pairs of centroid paths. The $O(\log n)$ cost that the algorithm in [FPT95a] incurs in processing each such interesting pair of paths arises when there are large (polynomial in n size) instances of the generalized Longest Increasing Subsequence Problem. At first sight, it is not clear that large instances of these problems can be created for sufficiently many of the interesting pairs; unfortunately, this turns out to be the case. However, these problem instances still have some useful structure. By using (static) weighted trees we process pairs of interesting vertices in $O(1)$ time per pair, on the average, as is shown by an appropriately parameterized analysis.

The paper is organized as follows. Section 2 gives some basic definitions and primitives. Section 3 outlines the algorithm. Section 4 provides further details of the algorithm and Section 5 gives the analysis. The remaining sections deal with problems raised in Sections 3-5.

2 Definitions and Preliminaries

All trees henceforth refer to binary trees whose internal nodes have exactly two children.

The tree $T(x)$ denotes the subtree of T rooted at vertex x. The *size* of a tree T, denoted by $|T|$, is the number of leaves in it. In our problem, $|T_1| = |T_2| = n$.

Given a binary tree T, its *centroid decomposition* is a partition of its vertices into disjoint paths obtained as follows. First, a path from the root of T to a leaf is traced in the following way: at each step the path is extended to that child of the current node x, whose subtree has at least half the nodes of the subtree rooted at x (ties are broken arbitrarily). Second, all the nodes on the above path are removed. This creates a forest of subtrees, each of which has at most

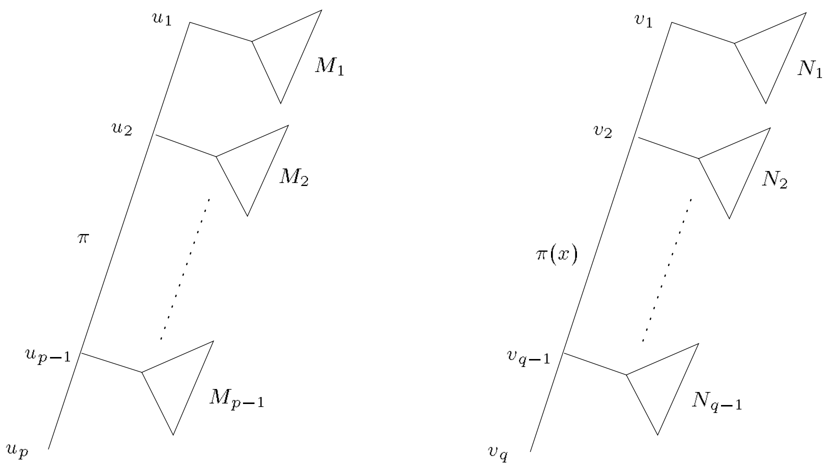

Figure 1: Paths π and $\pi(x)$.

$\frac{|T|}{2}$ nodes; the above procedure is applied recursively to each of these subtrees. Each path obtained as above is called a *centroid* path. Note that the centroid decomposition of T can easily be found in $O(n)$ time.

A tree T can be preprocessed in $O(|T|)$ time so that given any subset L of its leaves in left to right order, the subtree induced by L can be computed in $O(|L|)$ time. The details of this procedure are described in Section 8.

The set of labels at the leaves of T_1 is identical to that at the leaves of T_2. For a leaf l in one of these trees, the leaf with the same label in the other tree is called its *twin*. Two subtrees, one from each tree, are said to *intersect* if and only if some leaf in one subtree has a twin in the other. The subtree of T_2 induced by some subset of the leaves of T_1 is the subtree of T_2 induced by the twins of these leaves of T_1.

3 Algorithm Outline

Let $Agree(T_1, T_2)$ denote our algorithm for *MAST*, given trees T_1, T_2. For each vertex w in T_2, $Agree(T_1, T_2)$ computes the maximum agreement subtree of T_1 and $T_2(w)$.

Algorithm Outline. $Agree(T_1, T_2)$ has three steps.

Step 1. Compute the centroid decompositions of T_1 and T_2. This takes $O(n)$ time.

We need some definitions to describe the remaining steps.

Definitions. Let π be the centroid path containing the root of T_1. Let $p = |\pi|$ and $u_1, u_2, \ldots, u_{p-1}, u_p$ be the vertices on this path in order from the root. Let $M_1, M_2, \ldots, M_{p-1}$ comprise the forest of subtrees created by the removal of π from T_1. Let $m_i = |M_i|$, the number of leaves in M_i (see Fig.1). Recall that $m_i \leq n/2$ as π is a centroid path. For technical reasons, we define $m_p = 1$. Therefore, $\sum_{i=1}^{p} m_i = n$.

Step 2. For each i, $1 \leq i \leq p-1$, $Agree(M_i, S_i)$ is computed, where S_i is the subtree of T_2 induced by the leaves of M_i. S_i is determined in $O(m_i)$ time after the twins of the leaves of M_i have been ordered. The latter is easily done for all i together in $O(\sum_1^p m_i)$ time (see Section 8); for this, we charge $O(m_i)$ to tree M_i. The above along with the recursive call will take $O(m_i \log m_i)$ time for each i. For each vertex w in S_i, Step 2 computes the maximum agreement subtree of M_i and the subtree rooted at w. Note that if the maximum agreement subtree of T_1 and T_2 contains no vertex from π then it will be found in Step 2. Step 3 handles the other case.

Step 3: Matching π. For each $w \in T_2$, the largest agreement subtree for the trees T_1 and $T_2(w)$ is found. Informally, we call this the process of "matching" π at each of the vertices w of T_2. We will show how this is done in $O(\sum_{i=1}^{p} m_i \log \frac{n}{m_i} + n)$ time.

Clearly, the total time over all three steps is $O(n \log n)$. The following sections show how Step 3 is performed in $O(\sum_{i=1}^{p} m_i \log \frac{n}{m_i} + n)$ time. We need the following definitions to proceed further.

Definitions. Recall that the centroid decomposition of T_2 partitions its vertices into disjoint paths. The *beginning* of such a path is defined to be the vertex closest to the root of T_2 in that path. Let X denote the set of vertices in T_2 at which paths in the above decomposition begin.

We will define some bipartite graphs next. Each graph will correspond to a particular centroid path in the centroid decomposition of T_2 and will be used to match π at all the vertices in this centroid path. These graphs will have the property that a particular kind of matching, which can be computed efficiently, will correspond to agreement subtrees.

4 The Matching Graphs $G(x)$ and the π Matching Algorithm

We define a number of bipartite graphs, one for each $x \in X$. The graph $G(x)$ corresponding to vertex x is defined as follows.

Vertices of $G(x)$. The left vertex set $L(x)$ of $G(x)$ is a subset of $\{u_1, \ldots, u_p\}$. Vertex u_i, $1 \leq i \leq p-1$, is in the set if and only if M_i and $T_2(x)$ intersect. Vertex u_p is in the set if and only if its twin is in $T_2(x)$.

The right vertex set $R(x)$ of $G(x)$ is exactly the set of vertices in the centroid path beginning at vertex x.

Since both sets of vertices are drawn from centroid paths, we order the vertices on each side in the order they occur on their respective centroid paths. The *topmost* vertex is the closest and the *bottommost* is the furthest. Further, two edges (a, b) and (a', b') in $G(x)$ are said to *cross* if a is above a' and b is below b', or vice versa. In addition, edge (a, b) is said to *dominate* (a', b') in $G(x)$ if a is above a' and b is above b'. The *topmost* edge in a set of edges, if any, is the edge which dominates all other edges in that set.

Before defining the edges of $G(x)$, we need the following definitions.

Definitions. Let $\pi(x)$ be the centroid path containing the root of $T_2(x)$. Let q be the length of this path. Let $v_1, v_2, \ldots, v_q$ be the vertices on this path in order from the root. Let $N_1, N_2, \ldots, N_{q-1}$ comprise the forest of subtrees of $T_2(x)$ created by the removal of $v_1, \ldots, v_q$ from $T_2(x)$. Let $n_i = |N_i|$, for $i = 1 \ldots q-1$ (see Fig.1). For technical reasons, we define $n_q = 1$. Then $\sum_{i=1}^{q} n_i = |T(x)|$.

Edges of $G(x)$. $G(x)$ is actually a multigraph, where each multiedge consists of three edges, a *white* edge, a *red* edge and a *green* edge, each of which has a distinct weight associated with it.

A multiedge between $u_i \in L(x)$, $1 \leq i \leq p-1$ and $v_j \in R(x)$, $1 \leq j \leq q-1$, exists if and only if M_i and N_j intersect. The white edge in this multiedge has weight equal to the size of the maximum agreement subtree of M_i and N_j. The red edge in this multiedge has weight equal to the size of the maximum agreement subtree of $T_1(u_i)$ and N_j. The green edge in this multiedge has weight equal to the size of the maximum agreement subtree of M_i and $T_2(v_j)$. If $u_p \in L(x)$ then there is a multiedge between u_p and v_j such that either $j \neq q$ and u_p's twin is in N_j or $j = q$ and u_p's twin is v_q; all three edges in this multiedge have weight 1. In addition, there is a multiedge between u_i and v_q such that either $i \neq p$ and v_q's twin is in M_i or $i = p$ and v_q's twin is u_p; all three edges in this multiedge have weight 1.

Definition. We define a *proper crossing* in $G(x)$ to be a red-green edge pair in $G(x)$ such that the two edges cross and further, the endpoint of the green edge in $L(x)$ is above that of the red edge.

An Agreement Matching in $G(x)$. A matching in $G(x)$ is an agreement matching if:

1. It has zero or more white edges and at most one proper crossing.

2. No white edge crosses any other edge; further, all white edges dominate the edges in the proper crossing, if any.

See Fig.2. The weight of such a matching is just the sum of the weights of its edges. The following property of agreement matchings in G is crucial.

The Key Property. Each agreement matching corresponds to a structurally unique agreement subtree of T_1 and T_2, as made precise below. Consider an agreement matching $\mathcal{M}$ comprising white edges $(u_{i_1}, v_{j_1}), \ldots, (u_{i_k}, v_{j_k})$ followed by green edge $(u_{i'}, v_{j'})$ crossing red edge $(u_{i''}, v_{j''})$ (see Fig.2, where each subtree is the maximum agreement subtree of the indicated pair of subtrees). Also consider the tree $\mathcal{T}$ defined as follows. $\mathcal{T}$ has a path comprising the vertices $u_{i_1}, \ldots, u_{i_k}$ in sequence. The roots of maximum agreement subtrees of the tree pairs $(M_{i_1}, N_{j_1}), \ldots, (M_{i_k}, N_{j_k})$ are children of the vertices in the above path in this sequence. Finally, there is an additional vertex with edges to u_{i_k} and to the root of a maximum agreement subtree of the tree pairs $(M_{i'}, T_2(v_{j'})), (T_1(u_{i''}), N_{j''})$. Note that in general the maximum agreement subtrees need not be unique.

It is easily seen that the weight of $\mathcal{M}$ is the same as the size of $\mathcal{T}$. Also, it is clear that we can go in the opposite direction, converting any agreement tree to an agreement matching. Thus, in order to determine the maximum agreement subtree of T_1 and $T_2(x)$, it suffices to determine the maximum weight agreement matching in $G(x)$.

To describe the algorithm for matching π, we need the following definitions followed by an important theorem. The theorem itself will be proved in Section 7.

Definitions. The *degree* of a vertex in $G(x)$ is defined as the number of white edges incident on it. Let $d_i(x)$ denote the degree of u_i. A vertex in $L(x)$ is called a *singleton* vertex if it has degree one; the white edge incident on it is called a *singleton* edge. Let $nswe(x)$ denote the number of non-singleton white edges in $G(x)$. Let $nsav(x)$ denote the number of vertices in $R(x)$ which have at least one incident non-singleton white edge. Let $SV(x)$ denote the set of singleton vertices in $L(x)$.

Theorem 1 *Consider a particular $x \in X$. For each $u_i \in L(x)$, the largest weight agreement matching in $G(x)$ containing only edges incident on or below u_i in $L(x)$ can be found in time $O(\sum_{i|d_i(x)>1} d_i(x) \log \frac{nsav(x)}{d_i(x)} + \sum_{(u_i,v_j)|d_i(x)=1} \log \frac{|T(x)|}{n_j})$ time. Further, for each $v_j \in R(x)$, the largest weight agreement matching in $G(x)$ containing only edges incident on or below v_j in $R(x)$ can also be found in the same time.*

Theorem 1 is achieved by storing the vertices of $R(x)$ in an appropriately weighted search tree, described in Section 7.

Algorithm Outline for Matching π. The matching graphs $G(x)$ for all $x \in X$ will be constructed in time proportional to the sum of the sizes of these graphs.

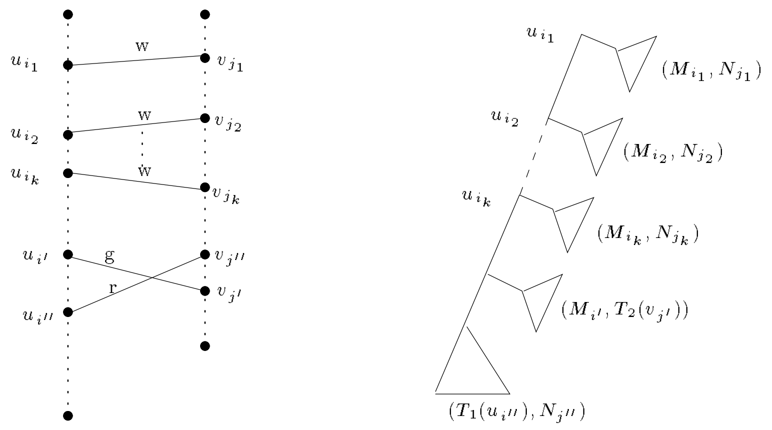

Figure 2: An Agreement Matching with the Associated Agreement Tree.

This construction is described in Section 6. Then each matching graph $G(x)$ is processed as follows (see Theorem 1). For each $u_i \in L(x)$, the largest weight agreement matching in $G(x)$ containing only edges incident on or below u_i in $L(x)$ is found. Further, for each $v_j \in R(x)$, the largest weight agreement matching in $G(x)$ containing only edges incident on or below v_j in $R(x)$ is also computed. This computation of agreement matchings is described in Section 7. For each $w \in T_2$, the largest agreement subtree of T_1 and $T_2(w)$ can be determined easily from the above information as it is given by the largest weight agreement matching in $G(x)$ comprising only edges incident upon or below vertex w in $R(x)$. Section 5 shows that the total time taken above is $O(\sum_{i=1}^{p} m_i \log \frac{n}{m_i})$, as required.

Inferring Maximum Agreement Subtrees. Consider a vertex $w \in T_2$; let x be the beginning of the centroid path in T_2 containing w. Then $w \in R(x)$. The maximum agreement subtree of T_1 and $T_2(w)$ is given by the largest weight agreement matching in $G(x)$ comprising only edges incident upon or below vertex w in $R(x)$.

5 The Analysis

We need the following preliminary lemmas before beginning the analysis.

Lemma 2 *Consider graph $G(x)$. Then*

$$\sum_{i|d_i(x)>1} d_i(x)\log\frac{nsav(x)}{d_i(x)} \leq \sum_{i|d_i(x)>1} d_i(x)\log\frac{n}{m_i}.$$

Proof: Multiplying each side by $\ln 2$, we get the following equivalent inequality:

$$A = \sum_{i|d_i(x)>1} d_i(x)\ln\frac{nsav(x)}{d_i(x)} \leq \sum_{i|d_i(x)>1} d_i(x)\ln\frac{n}{m_i} = B.$$

Let $\sum_{i|d_i(x)>1} d_i(x) = nswe(x)$. Let $\alpha_i(x) > 0$ be such that $d_i(x) = \alpha_i(x)\frac{m_i}{n}nswe(x)$. Then $\sum_{i|d_i(x)>1}\alpha_i(x)m_i = n$. Also note that $nsav(x) \leq nswe(x)$. Therefore, $A \leq B - \sum_{i|d_i(x)>1}\alpha_i(x)\frac{m_i}{n}nswe(x)\ln\alpha_i(x)$. It suffices to show that $C = \sum_{i|d_i(x)>1}\alpha_i(x)m_i\ln\alpha_i(x) > 0$.

We split C into two terms, $C_1 = \sum_{i|d_i(x)>1,\alpha_i(x)\geq 1}\alpha_i(x)m_i\ln\alpha_i(x)$ and $C_2 = \sum_{i|d_i(x)>1,0<\alpha_i(x)<1}\alpha_i(x)m_i\ln\alpha_i(x)$. $C_1 \geq \sum_{i|d_i(x)>1,\alpha_i(x)\geq 1}(\alpha_i(x) - 1)m_i$. Further, $\sum_{i|d_i(x)>1,\alpha_i(x)\geq 1}(\alpha_i(x) - 1)m_i - \sum_{i|d_i(x)>1,\alpha_i(x)<1}(1 - \alpha_i(x))m_i = \sum_{i|d_i(x)>1}(\alpha_i(x) - 1)m_i = \sum_{i|d_i(x)>1}\alpha_i(x)m_i - \sum_{i|d_i(x)>1}m_i \geq n - n \geq 0$.

Therefore $C \geq \sum_{i|d_i(x)>1,0<\alpha_i(x)<1}(1 - \alpha_i(x) + \alpha_i(x)\ln\alpha_i(x))m_i \geq 0$. ■

Definition. Let S_i denote the subtree of T_2 induced by the leaves of M_i.

Lemma 3 $\sum_{x\in X|d_i(x)>1} d_i(x) = O(m_i)$.

Proof: Consider $G(x)$ such that $d_i(x) > 1$. Then all but one of the vertices of $R(x)$ adjacent to u_i are also in S_i; this is because M_i intersects both the right and the left subtrees of all but the bottommost of the vertices adjacent to u_i in $R(x)$. Since each vertex in S_i is in at most one matching graph $G(x)$ and since $|S_i| = m_i$, the lemma follows. ■

.

Consider a vertex $u_i \in \pi$. From Theorem 1 and Lemma 2, the following work is assigned to u_i when considering matching graph $G(x)$, $x \in X$.

1. If M_i and $T_2(x)$ do not intersect then no work is assigned to u_i as u_i is not in $G(x)$.
2. If $d_i(x) = 1$ then the work assigned to u_i is $O(\log\frac{|T_2(x)|}{n_j})$.
3. If $d_i(x) > 1$ then the work assigned to u_i is $O(d_i(x)\log\frac{n}{m_i})$.

The following is a corollary of Lemma 3 and the above bounds.

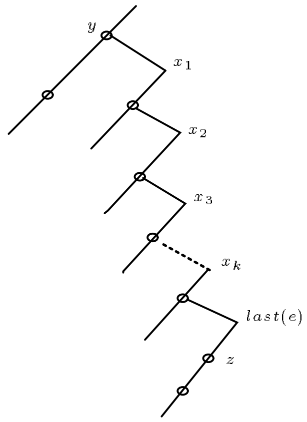

Figure 3: Portion of T_2 showing vertices in $H(e)$ and $last(e)$.

Corollary 4 *The work assigned to vertex u_i over all matching graphs $G(x)$ with $d_i(x) > 1$ is $O(m_i \log \frac{n}{m_i})$.*

It now suffices to account for the work assigned to vertex u_i over all matching graphs $G(x)$ with $d_i(x) = 1$. Next, we show that this work is also $O(m_i \log \frac{n}{m_i})$. We use the tree S_i for this analysis.

Analyzing over S_i. Note that for each $x \in X$ such that $d_i(x) = 1$, x is not in S_i, i.e., it lies on the path in T_2 between the endpoints of some edge in S_i; x is said to *lie* on this edge of S_i.

Consider the maximal subset $H(e)$ of vertices x in X which lie on edge e of S_i and for which $d_i(x) = 1$. Let $H(e) = \{x_1, x_2, \ldots, x_k\}$; here the vertices appear in increasing order of distance from the root of T_2. Let $e = (y, z)$, with y the parent of z in S_i. Let $first(e) = x_1$. Let $last(e)$ be the first vertex x in X such that $d_i(x) > 1$ and x is on the path from x_k to z in T_2, if any; otherwise let $last(e)$ be z (and then z is a leaf). See Fig.3.

Lemma 5 *The sum of $|T_2(first(e))|$ over any subset of edges of S_i, no two of which lie on the same root-to-leaf path in S_i, is $O(n)$.*

Proof: The above subtrees of T_2 are all disjoint. ∎

.

Lemma 6 *The work assigned to u_i on edge e, i.e., in processing graphs $G(x)$, $x \in H(e)$, is $O(\log \frac{|T_2(first(e))|}{|T_2(last(e))|})$.*

Proof: The work assigned to u_i in processing $G(x_j)$ is $O(\log \frac{|T_2(x_j)|}{|T_2(x_{j+1})|})$ for $1 \leq j < k$ and $O(\log \frac{|T_2(x_k)|}{|T_2(last(e))|})$ for $j = k$. Thus the sum of the work assigned to u_i at the graphs $G(x_j)$ is $O(\log \frac{|T_2(first(e))|}{|T_2(last(e))|})$. ■

Lemma 7 *Consider edges $e, e' \in S_i$ such that e connects vertex z to its parent and e' connects vertex z to one of its children. Then $|T_2(last(e))| \geq |T_2(first(e'))|$.*

Proof: $first(e')$ is a descendant of $last(e)$ in T_2. ■

We claim that sum of the work assigned to u_i over all the edges of S_i is $O(m_i \log \frac{n}{m_i})$. We show this next by applying tree contraction on S_i.

Removal step. First, remove all edges in S_i incident upon leaves in S_i. The work done on these edges is bounded by the sum over all such edges e of $O(\log |T_2(first(e))|)$; further, the number of such edges is at most m_i. By Lemma 5, this sum is at most $O(m_i \log \frac{n}{m_i})$.

Contract step. Next, contract all paths consisting only of degree two vertices in S_i into a single edge. The H set for such an edge e is defined to be the union of the H sets for the edges comprising the path which was contracted to give e. The work done on e is also defined to be the sum of the work done on the relevant edges. $first(e)$ and $last(e)$ are again defined as before. As is easily seen, Lemma 5, Lemma 6, and Lemma 7 hold for the new contracted tree as well. Further, this new tree has at most $m_i/2$ leaves.

Wrapping Up. $O(\log m_i)$ phases of the removal and contract steps are performed. In the jth phase, the work done on the edges removed is $O(\frac{m_i}{2^{j-1}} \log \frac{n 2^{j-1}}{m_i})$. Summing up over all phases, we get

Lemma 8 *The work assigned to vertex u_i over all matching graphs $G(x)$ with $d_i(x) = 1$ is $O(m_i \log \frac{n}{m_i})$.*

The following lemma is needed in the next section.

Lemma 9 *The total number of edges incident to u_i over all matching graphs is $O(m_i \log \frac{n}{m_i})$.*

Proof: Recall that the work attributed to a singleton edge is of the form $\log \frac{|T_2(x)|}{n_j} \geq 1$. Thus, Lemma 8 implies that there are $O(m_i \log \frac{n}{m_i})$ singleton edges incident to u_i, and by Lemma 3, there are $O(m_i)$ non-singleton edges incident to u_i. ■

As discussed in Section 3, Corollary 4 and Lemma 8 imply:

Theorem 10 *There is an algorithm for the Maximum Agreement Subtree Problem for two binary trees with an $O(n \log n)$ running time.*

6 Constructing the Matching Graphs

We show how all the matching graphs can be set up in time proportional to the sum of their sizes, which by Lemma 9 is $O(\sum_{i=1}^{p-1} m_i \log \frac{n}{m_i})$. First, we show how to set up the vertices and edges in each graph. Then we show how the weights on the edges are computed.

Preprocessing. T_2 is preprocessed in linear time to compute a pointer from each vertex to the beginning of the centroid path containing it. It is also preprocessed to enable induced subtree computations in the same time bounds.

6.1 Setting up Vertices and Edges.

The matching graphs in which vertex u_i appears along with the multiedges incident upon it in these graphs are determined in time proportional to the sum of the number of such multiedges over all such graphs as follows.

Processing u_p. First, consider the leaf u_p of T_1. The only matching graphs containing u_p are those which correspond to centroid paths beginning at vertices x of T_2 such that x is an ancestor of the twin of u_p in T_2. Further, if $u_p \in L(x)$ then there is a multiedge between u_p and vertex $y \in R(x)$ if and only if y is the nearest ancestor of u_p's twin in the centroid path beginning at x. Thus the matching graphs to which u_p belongs and the multiedges incident on u_p in these graphs can be determined in time proportional to the number of such graphs, given pointers from each vertex in T_2 to the beginning of the centroid path containing it.

Lemmas 11 and 12 are needed for the next step.

Lemma 11 *If vertex v_j in the centroid path beginning at vertex x of T_2 is in S_i then u_i is adjacent to v_j in $G(x)$.*

Proof: Clearly, if $j \neq q$ then M_i and N_j intersect and if $j = q$ then v_j's twin is in M_i. ∎

Lemma 12 *If vertex v_j in the centroid path beginning at vertex x of T_2 is not in S_i then u_i is adjacent to v_j in $G(x)$ if and only if $v_j \neq v_q$ and there exists some vertex $y \in S_i$ which is in N_j.*

Proof: We assume that $j \neq q$. For if $j = q$ then $v_j = v_q$ is a leaf of T_2 and since it does not appear in S_i, its twin is not in M_i, and therefore, there is no edge between u_i and v_j.

First, suppose some vertex $y \in S_i$ is in N_j. Then, clearly, N_j intersects M_i. Therefore, there must be an edge between u_i and v_j in $G(x)$. Next, suppose that there is such an edge. Then N_j intersects M_i. Therefore, there exists some $y \in S_i$ which is in N_j. ∎

Processing $u_i, 1 \leq i \leq p-1$**.** The subtrees of T_2 induced by leaves of each M_i are computed in $O(\sum_1^{p-1} m_i)$ time as described in Section 8. Let S_i denote this induced subtree. For each vertex z in S_i, perform the following in T_2 until a vertex in the centroid path containing the parent of z in S_i is reached: repeatedly jump to the parent of the beginning of the centroid path in T_2 containing the current vertex. By Lemmas 11 and 12, there is a multiedge from u_i to each vertex y of T_2 (in the corresponding matching graph containing y) encountered in this following procedure. Thus this procedure takes time proportional to the sum of the number of multiedges incident on u_i over all matching graphs it lies in, given pointers from each vertex in T_2 to the beginning of the centroid path containing it.

Remark. For an edge between u_i and v_j, $i \neq p$, $j \neq q$, $v_j \notin S_i$, the vertex $y \in S_i$ such that v_j is on the path from y to z in T_2 can be easily computed in the course of the above procedure, where z is the parent of y in S_i. We define $map(i,j)$ to be the vertex y as above if $v_j \notin S_i$ and the vertex v_j itself, otherwise.

6.2 Determining Edge Weights in $G(x)$.

Recall that for a multiedge between u_i and v_j in $G(x)$, we need to determine the sizes of the maximum agreement subtrees of the following pairs of trees.

1. M_i, N_j: white edge weight.
2. $T_1(u_i)$, N_j: red edge weight.
3. M_i, $T_2(v_j)$: green edge weight.

Also recall that the multiedge itself indicates that M_i and N_j intersect.

Assume that the agreement matchings in graphs $G(x')$ have already been determined, where x' is a descendant of x in T_2. Using this information and the information computed in Step 2, we show how the above required information can be computed for multiedge (u_i, v_j) in graph $G(x)$ in constant time. Recall that in Step 2, the maximum agreement subtrees of M_i and the subtrees rooted at each vertex w of S_i were determined.

White Edge Weight. Let $y = map(i,j)$. If $y \neq v_j$ then the maximum agreement subtree of M_i and the subtree of S_i rooted at y gives the desired information. Suppose $y = v_j$, i.e, $y \in S_i$. Let z be the child of $y \in S_i$ such that $z \in N_j$. The maximum agreement subtree of M_i and the subtree of S_i rooted at z gives the desired information in this case. This takes constant time.

Green Edge Weight. The maximum agreement subtree of M_i and the subtree of S_i rooted at $y = map(i,j)$ gives the desired information in constant time.

Red Edge Weight. Let y be the root of N_j. Recall that the agreement matchings in graphs $G(x')$ have already been determined, where $x' \in X$ is a descendant of x in T_2. Since $y \in X$, agreement matchings in graph $G(y)$ would already have been computed. Recall from Theorem 1 that for each vertex in $L(y)$, the maximum weight agreement matching containing only edges incident on or below that vertex in $L(y)$ has been computed.

Note that since M_i intersects with $T_2(y)$ (since a multiedge exists between u_i and v_j) $u_i \in L(y)$. The largest weight agreement matching in $G(y)$ containing only edges incident on or below vertex u_i in $L(y)$ gives the desired information. This information is computed as graph $G(y)$ was processed, so it can be accessed in constant time now.

7 Computing Agreement Matchings

Consider graph $G(x)$. Recall that for each vertex in $L(x)$, we need to compute the largest weight agreement matching containing only edges incident on or below it in $L(x)$, and likewise for each vertex in $R(x)$. We outline the algorithm before giving details. The algorithm is similar to that in [FPT95a], but the data structure we use and the associated operations are different.

Algorithm Outline. First, a weight balanced binary search tree $\mathcal{T}$ whose leaves are the vertices in $R(x)$ is set up; here, the vertices in $R(x)$ are given appropriate weights yet to be described. Next, the vertices in $L(x)$ are considered in turn in bottom-to-top order. For each vertex $u_i \in L(x)$, the vertices adjacent to it in $R(x)$ are searched for in $\mathcal{T}$; the largest weight agreement matching with each white edge incident on u_i as topmost edge is found in the course of this search, as is the largest weight proper crossing for each green edge incident on u_i. From the above information, the largest weight agreement matching containing only edges incident on or below u_i in $L(x)$ is easily found. Following the above search, the information stored in $\mathcal{T}$ is updated. The time taken for processing u_i will be $O(d_i(x) \log \frac{nsav(x)}{d_i(x)})$ if $d_i(x) > 1$ and $O(\log \frac{|T(x)|}{n_j})$ if $d_i(x) = 1$ and u_i is adjacent to v_j in $G(x)$. After all vertices in $L(x)$ have been processed, the vertices in $R(x)$ are processed. For all such vertices v_j, the largest weight agreement matching containing only edges incident on or below v_j in $R(x)$ are found in $O(|R(x)|)$ time by a single scan of $\mathcal{T}$. The bounds in Theorem 1 follow.

Weighted Search Tree $\mathcal{T}$. Vertex $v_j \in R(x)$ is given weight $n_j + \frac{|T(x)|}{nsav(x)}$ if some non-singleton edge in $G(x)$ is incident upon it, and weight n_j, otherwise. The sum of the weights of vertices in $R(x)$ is at most $2|T(x)|$. The construction of $\mathcal{T}$ using these weights is in Section 9.

Tree $\mathcal{T}$ has the following three crucial characteristics.

1. $\mathcal{T}$ can be constructed in $O(|R(x)|)$ time.

2. Searching for v_j in $\mathcal{T}$ takes $O(\log \frac{|T(x)|}{n_j})$ time.

3. Searching for an ordered subset $\{v_{j_1}, \ldots, v_{j_k}\}$ of $R(x)$, each vertex in which has an incident non-singleton edge, takes $O(k \log \frac{nsav(x)}{k})$ time. The procedure used here is to first search for v_{j_1} starting at the root, then search for v_{j_2} starting at v_{j_1} in the obvious way, and so on.

Auxiliary Information in $\mathcal{T}$. We maintain the following auxiliary information at each internal vertex in $\mathcal{T}$. Recall that we process the vertices of $L(x)$ in order; an edge of $G(x)$ is said to be *in* $\mathcal{T}$ if its endpoint in $L(x)$ has already been processed. Further, we say that an edge of $G(x)$ is *in* $\mathcal{T}(z)$ if its endpoint in $L(x)$ has been processed and is located in $\mathcal{T}(z)$. Let $anc(z)$ denote the set of ancestors of z in $\mathcal{T}$, z inclusive. For a leaf $v_j \in \mathcal{T}$, $lfringe(v_j)$ is the set of vertices in $\mathcal{T}$ which are left children of vertices in $anc(v_j)$. $rfringe(v_j)$ is defined analogously.

The following information is maintained at each vertex z of $\mathcal{T}$.

1. $g(z)$: For each z, $\max_{z' \in anc(z)} g(z')$ will be the heaviest green edge in $\mathcal{T}$ which forms a proper crossing with each red edge in $\mathcal{T}(z)$.

2. $x(z)$: This is largest weight proper crossing among the edges in $\mathcal{T}(z)$.

3. $m(z)$: This is largest weight agreement matching containing a white edge such that the topmost white edge is in $\mathcal{T}(z)$.

4. $y(z)$: This the largest weight proper crossing such that the green edge in this crossing is in $\mathcal{T}$ but not in $\mathcal{T}(z)$, the red edge in this crossing is in $\mathcal{T}(z)$, and the green edge does not form a proper crossing with all the red edges in $\mathcal{T}(z)$.

5. $r(z)$: This is simply the heaviest red edge in $\mathcal{T}(z)$.

Next, we show how vertex u_i is processed, given that vertices below it in $L(x)$ have been processed. For the moment assume that $d_i(x) = 1$. The case when $d_i(x) > 1$ will be addressed later.

Case 1. $d_i(x) = 1$. Let v_j be the only vertex to which u_i is adjacent. First, v_j is found in $\mathcal{T}$; this takes $O(\log \frac{|T(x)|}{n_j})$ time. Next, the white, red, and green edges incident on u_i are processed as described below in the same time bound. An important fact to note is that in each case, the information in $\mathcal{T}$ will be read and updated only at vertices in the set $anc(v_j)$ and vertices which are children of vertices in this set; the time taken in this process will be proportional to the depth of v_j, i.e., $O(\log \frac{|T(x)|}{n_j})$.

Processing White Edge $e = (u_i, v_j)$. First, the largest weight agreement matching with e as the topmost edge is determined. Then the $m()$ values at vertices in $anc(v_j)$ are updated according to the weight of this matching. All other information remains unchanged.

The above desired matching is computed as follows. There are two cases. Either this matching contains another white edge. The largest weight matching among all such matchings is given by $1 + \max_{z \in lfringe(v_j)} m(z)$. The other case occurs when this matching contains only edge e plus a proper crossing. Thus, it suffices to compute the largest proper crossing containing edges dominated by e. This is given by $\max\{\max_{z \in lfringe(v_j)} x(z), \max_{z \in lfringe(v_j)} (\max_{z' \in anc(z)} g(z')) + r(z), \max_{z \in lfringe(v_j)} y(z)\}$. The first term here is the largest weight proper crossing in which both edges are in $\mathcal{T}(z)$ for some $z \in lfringe(v_j)$. The second term is the largest weight proper crossing in which the red edge is in $\mathcal{T}(z)$, for some $z \in lfringe(v_j)$, the green edge is not in this subtree but it forms a proper crossing with each red edge in this subtree. The third term is the largest weight proper crossing in which the red edge is in $\mathcal{T}(z)$, for some $z \in lfringe(v_j)$, the green edge is not in this subtree and it does not form a proper crossing with some red edge in this subtree.

Processing Red Edge $e = (u_i, v_j)$. The $m()$ and $x()$ values remain unchanged in $\mathcal{T}$. Next, note that no green edge already in $\mathcal{T}$ can form a proper crossing with e. This implies that the $y()$ and $g()$ values for $z \in anc(v_j)$ need to be modified.

Consider $y(z)$ first, $z \in anc(v_j)$. A green edge in $\mathcal{T}$ which formed a proper crossing with all red edges in $\mathcal{T}(z)$ does not do so any more. So $y(z)$ is set to $\max\{y(z), (\max_{z' \in anc(z)} g(z')) + r(z)\}$.

Consider $g(z)$ next, $z \in anc(v_j)$. $g(z)$ is set to ϕ. Before this is done, $g(y)$ is updated to $\max_{y' \in anc(y)} g(y')$ for each $y \in lfringe(v_j)$ and $y \in rfringe(v_j)$. The invariant on $g()$ is easily seen to be maintained.

Finally $r(z)$ is set to $\max\{r(z), wt(e)\}$, for each z, $z \in anc(v_j)$.

Processing Green Edge $e = (u_i, v_j)$. Note that e can form a proper crossing with only those red edges in $\mathcal{T}$ which are in $\mathcal{T}(z)$, $z \in rfringe(v_j)$; further, e forms a proper crossing with each such red edge. Therefore, $g(z)$ is set to $\max\{\max_{z' \in anc(z)} g(z'), wt(e)\}$ for each $z \in rfringe(v_j)$.

For each $z \in anc(v_j)$, $x(z)$ is then set to the larger of the current value and $\max(wt(e) + r(z'))$, the maximum being taken over all vertices $z' \in rfringe(v_j)$ which are descendants of z. Also note that $\max_{z \in rfringe(v_j)}(wt(e) + r(z))$ gives the largest weight proper crossing containing e.

Case 2: $d_i(x) = k > 1$. Suppose u_i is adjacent to $v_{j_1}, v_{j_2}, \ldots, v_{j_k}$, in bottom to top order. Then these vertices are searched for in sequence in $\mathcal{T}$. This takes $O(k \log \frac{nsav(x)}{k})$ time by the procedure mentioned earlier, i.e., first search for v_{j_1} starting at the root, then search for v_{j_2} starting at v_{j_1} in the obvious way, and so on. In the above process, all vertices in the set $\{z | z \in \{anc(v_{j_1}) \cup anc(v_{j_2}) \cup \ldots \cup anc(v_{j_k})\}\}$ are traversed. Again, as in Case 1, only information at vertices in the above set and at children of vertices in the above set needs to be read and updated. This takes time proportional to the size of the above set, which, in turn, is $O(k \log \frac{nsav(x)}{k})$.

Processing $R(x)$. It remains to show how, for each $v_j \in R(x)$, the largest weight agreement matching containing only edges incident on or below v_j in $R(x)$, is computed.

For each $v_j \in R(x)$, we find the largest weight agreement matching with some white edge incident upon v_j as the dominant edge and the largest weight proper crossing containing some red edge incident on v_j. This information clearly suffices. The first of the above two is given simply by $m(v_j)$. The second is given by $\max\{y(z), \max_{z' \in anc(v_j)} g(z') + r(v_j)\}$. Over all $v_j \in R(x)$, the computation of the above two values can be accomplished in a single pass of $\mathcal{T}$ in $O(|R(x)|)$ time.

8 Computing Induced Subtrees

We show how to preprocess a tree in $O(|T|)$ time so that given any subset L of its leaves in order, the subtree induced by L can be computed in $O(|L|)$ time. The construction is a generalization of the proof of Lemma 5.2 in [FT95].

T is preprocessed for *Least Common Ancestor* (LCA) queries in $O(|T|)$ time. This enables the computation of the LCA of any two leaves of T in constant time [HT84]. The distance of each vertex from the root of T is also computed; call this quantity the *depth* of a vertex.

Given the ordered set of leaves $L = l_1, l_2, \ldots, l_{|L|}$, the following steps are executed. First, the LCA l'_i of each pair of consecutive leaves l_i, l_{i+1}, $1 \leq i \leq |L| - 1$, is found; the l'_is will be the internal vertices in the subtree induced by L. Next, the edges between vertices are set up as follows. For each vertex v in the sequence $l_1, l'_1, l_2, l'_2, \ldots, l_{|L|-1}, l'_{|L|-1}, l_{|L|}$, two vertices v_{left} and v_{right} are computed. v_{left} is the nearest vertex to the left of v, if any, which has depth strictly less than v. v_{right} is defined analogously. This computation is easily accomplished in $O(|L|)$ time. Finally, edges are put between v and one of v_{left}, v_{right}, whichever has greater depth. The root of the induced tree will be the unique vertex for which both v_{left} and v_{right} are undefined.

Step 2 of the Main Algorithm. Step 2 (see Section 3) requires finding the induced trees S_i for each M_i, $1 \leq i \leq p-1$, in $O(\sum_i^{p-1} m_i)$ time. This is done in two steps (essentially this procedure is described in [FT95]). First, the leaves of each M_i are sorted by the order in which their twins occur in T_2. This is done by bucket sorting all the leaves of T_1 by the order in which their twins occur in T_2, and then bucket sorting them (in a stable way) by the order in which the trees M_i to which they belong occur in T_1. This takes $O(\sum_i^{p-1} m_i)$ time.

Next, for each M_i, $1 \leq i \leq p-1$, the subtree of T_2 induced by the leaves of M_i is found using the algorithm described above in $O(m_i)$ time. The total time taken is $O(\sum_i^{p-1} m_i)$.

9 The Weighted Search Tree

Recall that we are given vertices $v_1, v_2, \ldots, v_{|R(x)|}$, such that vertex v_j has weight $w(v_j) = n_j + \frac{|T(x)|}{nsav(x)}$ if it has an incident non-singleton edge, and weight $w(v_j) = n_j$, otherwise.

We construct a weight balanced tree $\mathcal{T}$ in $O(|R(x)|)$ time. The weight $w(z)$ of an internal node z is the sum of the weights of the leaves in its subtree. In this construction, the weight of the grandparent of an internal node v, if it has one, is at least twice the weight of v, as is shown in Lemma 13. The construction follows.

If there is just one vertex it forms the tree. Otherwise, the vertices are partitioned into two groups, so that the total weight is as evenly partitioned as possible. To explain the precise rule, the following notation is helpful. $diff_j = |\sum_{i=1}^{j} w(v_i) - \sum_{i=j+1}^{|R(x)|} w(v_i)|$. The vertices are partitioned into groups $v_1, \cdots, v_j$ and $v_{j+1}, \cdots, v_{|R(x)|}$, such that $diff_j \leq diff_{j+1}, diff_{j-1}$ (ties are broken arbitrarily). Clearly, a binary search can locate j. A binary search that grows from each of the two endpoints of the ordered collection of vertices can find j in time $O(\log \min\{j, |R(x) - j\}$. Then the two collections are recusively formed into binary trees, and provide the left and right subtrees, respectively, for the complete tree. It is not hard to show that the construction of this tree takes time $O(|R(x)|$ overall.

Lemma 13 *Let z be an internal node of the tree, and suppose it has a grandparent z''. Then $w(z'') > 2w(z)$.*

Proof: Let z' be the parent of z. WLOG let z' be the left child of z''. Let v be the rightmost vertex in the subtree rooted at z'. Then $w(z') - \frac{1}{2}w(v) \leq \frac{1}{2}w(z'')$, for otherwise v would be placed in the right subtree of z''. As v must be in the right subtree of z', the left subtree of z' has weight less than $\frac{1}{2}w(z'')$. Thus we need only consider the case that z is the right child of z'. As z is not a leaf, its subtree contains at least two vertices. Let u be the leftmost such vertex. Then $w(z) - \frac{1}{2}w(u) \leq \frac{1}{2}w(z')$. Thus $2w(z) - w(u) - \frac{1}{2}w(v) \leq \frac{1}{2}w(z'')$. As $w(z) \geq w(u) + w(v)$, it follows that $w(z) < \frac{1}{2}w(z'')$. ■

It follows from Lemma 13 that the time to search for v_j in $\mathcal{T}$ is $O(1 + \log \frac{|2T(x)|}{w(v_j)}) = O(1 + \log \frac{|T(x)|}{n_j})$.

Next consider the case when an ordered subset $\{v_{j_1}, \ldots, v_{j_k}\}$ of vertices is given, each having an incident non-singleton edge. The algorithm to search for these vertices is to first start from the root and search for v_{j_1}, then start from v_{j_1} and search for v_{j_2}, then start from v_{j_2} and search for v_{j_3}, and so on. Each search is performed in the obvious way. For, technical reasons, we return to the root at the end.

Each edge in $\mathcal{T}$ is traversed at most twice during the above search, once in each direction.

Consider the topological subtree formed by the traversed edges. It has k leaves and $k-1$ internal nodes with two children. These nodes form a tree, with each edge in the tree corresponding to a path in the search tree. We associate with a node in the topological subtree the path in the search tree corresponding to the edge from the node to its parent in the topological subtree. The associated path for the root node of the topological subtree is the path from this node to the root of the search tree. We will give an upper bound on the total number of vertices on these paths excluding their endpoints. To do this, we give a lower bound on the total weight of the "off-path" subtrees for each path. (An "off-path" subtree for a node is simply the subtree which does not contain the continuation of the path.)

Let l be the number of internal vertices on one such path associated with node v. By Lemma 13, the sum of the weights of every second root of the off-path subtrees is at least $(2w(v)-w(v))+(4w(v)-2w(v))+\cdots+(2^{\lfloor l/2\rfloor}w(v)-^{\lfloor l/2\rfloor-1} w(v)) = (2^{\lfloor l/2\rfloor}-1)w(v)$. But $w(v) \geq \frac{|T(x)|}{nsav(x)}$ and the total weight is at most $2T(x)$. Simple calculus shows that the sum of these lower bounds on the path lengths is maximized when the terms $w(v)$ are all at their minimum value and the path weights are all equal at $2T(x)/(2k-1)$. This gives path lengths of $O(\log\frac{nsav(x)}{2k-1})$ and hence a total path length of $O(k\log\frac{nsav(x)}{k})$.

10 Concluding remarks

We can generalize our technique to higher degree bounds $d > 2$, by combining it with techniques from [FT95, Section 2] for unbounded degrees. This appears to yield an algorithm with running time $O(\min\{n\sqrt{d}\log^2 n, nd\log n\log d\})$. We conjecture, however, that there is an algorithm with running time $O(n\sqrt{d}\log n)$.

References

[AK94] A. Amir, D. Keselman. *Maximum agreement subtrees in multiple evolutionary trees.* SIAM J. Computing, to appear.

[CR96] R.. Cole, R. Hariharan. *An $O(n\log n)$ algorithm for the maximum agrement subtree problem for binary trees.* Proc. of the 7th ACM-SIAM SODA, pp. 323–332, 1996.

[FT94] M. Farach, M. Thorup. *Optimal evolutionary tree comparison by sparse dynamic programming.* Proc. of the 35th IEEE FOCS, pp. 770–779, 1994.

[FT95] M. Farach, M. Thorup. *Fast comparison of evolutionary trees.* Information and Computation, 123(1), pp. 29–37, 1995.

[FPT95a] M. Farach, T. Przytycka, M. Thorup. *The maximum agreement subtree problem for binary trees.* Proc. of 2nd ESA, 1995.

[FPT95b] M. Farach, T. Przytycka, M. Thorup. *Agreement of many bounded degree evolutionary trees.* Information Processing Letters, 55(6), pp. 297–301, 1995.

[FG85] C. R. Finden, A. D. Gordon. *Obtaining common pruned trees.* Journal of Classification, Vol. 2, pp. 255–276, 1985.

[GY95] R. Grishman, R. Yangarber. Private Communication, NYU, 1995.

[HT84] D. Harel and R.E. Tarjan. *Fast algorithms for finding nearest common ancestor.* Computer and System Science, 13, p. 338–355, 1984.

[Ka95] M-Y. Kao. *Tree contractions and evolutionary trees.* To appear in SIAM Jornal of Computing, 1995.

[KKM92] E. Kubicka, G. Kubicki, F. R. McMorris. *An algorithm to find agreement subtrees.* Journal of Classification, 1992.

[SW93] M. Steel, T. Warnow. *Kaikoura tree theorems: computing the maximum agreement subtree.* Information Processing Letters, 48, pp. 77–82, 1993.