On matching nodes between trees

Li Zhang

Systems Research Center
Hewlett-Packard Labs
1501 Page Mill Road
Palo Alto, CA 94304
l.zhang@hp.com

Abstract

Consider two rooted leaf-labeled trees T_1, T_2. Define the similarity between two internal nodes, one from each tree, to be $\frac{|A \cap B|}{|A \cup B|}$, where A, B are the sets of the leaves under the two nodes, respectively. In this paper, we consider the problem of computing for every node in T_1, the best matching node in T_2 under the above similarity measure. Such problem arises in applications such as comparing phylogenetic trees. In this paper, we present an $O((n \log n)^{1.5})$ time algorithm for the problem. The major difficulty in solving this problem is that the above similarity measure is non-linear while the traditional algorithms normally deal with linear weights. To overcome the difficulty, one novelty in our solution is to reduce the problem to computing the upper-envelope of pseudo-planes and then apply the results from Computational Geometry to obtain an efficient algorithm.

1 Introduction

Representing an object in a tree-like structure is ubiquitous in many areas. For examples, a phylogenetic (evolutionary) tree represents the evolution history of organisms; a DOM tree represents an XML document. In all such areas, it is important to be able to compare trees. One way for tree comparison is to define a distance function in the tree space to measure how different two trees are. In a more refined comparison, one usually establishes a mapping between similar nodes in two trees according to certain similarity measure. The mapping tells us more about where two trees are similar. In this paper, we study the problem of mapping nodes between two leaf-labeled trees. A tree is leaf-labeled if its leaves are labeled but the internal nodes are not. Such trees arise in the areas such as Classification, Biology. For example, a phylogenetic tree is a leaf-labeled tree.

In a leaf-labeled tree, the mapping between leaf nodes is immediate according to their labels. However, the mapping between internal nodes is not obvious. Intuitively, an internal node represents the cluster of its leaf set, the set of leaves under it. Therefore, one natural choice is to establish the mapping according to the similarity between the leaf sets. Indeed, the popular consensus tree approach [1, 10, 6, 14] computes the mapping from the node u in T_1 to the node v in T_2 if v has the same leaf set as u. One problem with the consensus method is that it only correlates two perfectly matching nodes, the pair of nodes with the identical leaf sets. For nodes which do not have perfectly matching nodes, the mapping is undefined even if two nodes have 99% overlap in their leaf sets. Stinebrickner [11, 12] proposed s-consensus trees to deal with this problem. In an s-consensus tree, the mapping is based on the best matching node instead of the perfectly matching node. The particular similarity measure proposed by Stinebrickner is induced from the *set similarity measure*: the similarity $S(A, B)$ between two sets A, B is defined to be $\frac{|A \cap B|}{|A \cup B|}$. One nice property of this measure is that the function defined by $d(A, B) = 1 - S(A, B)$ is a metric. This measure is also used for detecting similar documents in Stanford SCAM project [8] and by Broder *et al.* [3, 2].

As shown in [4], all the perfectly matching node pairs can be computed in the optimal linear time in terms of the number of tree nodes. It is easy to show that the best matching node of a single node can be computed in linear time, and therefore the mapping can be computed in quadratic time. In [3], an efficient algorithm is proposed to compute the highly similar document according to the set similarity measure. However, the algorithm is probabilistic and only works for finding out documents (or nodes) with high similarity score. In this paper, we show a deterministic algorithm that computes for every node in T_1, the best matching node in T_2 in roughly $O(n^{1.5})$ time where n denotes the number of leaves in T_1 and T_2. One difficulty in solving this problem is that the set similarity measure is non-linear. If we had defined the similarity as the size of intersection between two sets, then by taking advantage of the linearity, namely $|A \cap (B \cup C)| = |A \cap B| + |A \cap C|$ when

$B \cap C = \emptyset$, we would be able to compute all the best matching nodes in about linear time by using the classical link-cut tree [9, 13]. Unfortunately, the set similarity measure does not possess the same nice property. To overcome the difficulty, we reduce the problem to computing the maximum map of a set of function of the form $f_{a,b}(x,y) = \frac{a+x}{b+y}$. While those functions are non-linear, they behave similar to linear functions: the images of two such functions intersect at a simple open curve, and the images of three such functions intersect at a single point. The images of such functions are known as pseudo-planes in Computational Geometry [7]. By applying the techniques for computing the upper-envelope of pseudo-planes and combining with a structural property of trees, we are able to achieve an $O((n \log n)^{1.5})$ time algorithm.

The paper is organized as follows. In Section 2, we introduce some notations and useful properties. An immediate consequence of those properties is a simple algorithm for the best match problem that runs in quadratic time in the worst case and roughly linear time for balanced trees. In Section 3, we define a sub-problem called the ancestor match problem and show a solution for it. Then in Section 4 we present the $O((n \log n)^{1.5})$ algorithm for the best match problem by using the algorithm developed for the ancestor match problem.

2 Preliminaries

In what follows, we only consider binary trees. Extension of our algorithms to arbitrary trees is straightforward. Suppose that T is a rooted tree. We assume that T does not have any degree two nodes other than the root node. Let U be the universe of labels. T is leaf-labeled if every leaf of T is assigned a distinct label from U. For any node u in T, define its leaf set $L(u)$ to be the set of labels of the leaves in the subtree rooted at u. Denote by $L(T)$ the set of leaf labels in T. For two leaf-labeled trees T_1 and T_2, we define the similarity $S(u_1, u_2)$ between a node $u_1 \in T_1$ and a node $u_2 \in T_2$ as follows:

$$S(u_1, u_2) = \frac{|L(u_1) \cap L(u_2)|}{|L(u_1) \cup L(u_2)|}\,.$$

For the node $u_1 \in T_1$, the best match $M(u_1) \in T_2$ is the node that maximizes the similarity between u_1 and any node in T_2, i.e.

$$M(u_1) = \arg\max_{u_2 \in T} S(u_1, u_2)\,.$$

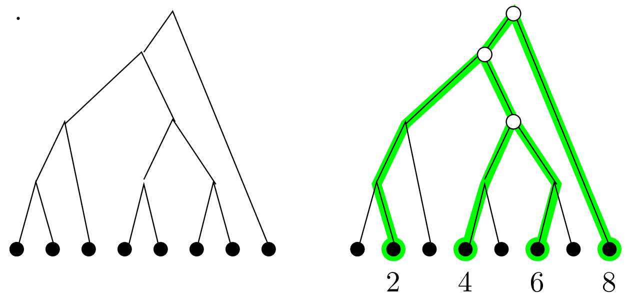

Figure 1. The spanning tree and simplified spanning tree of the set $\{2, 4, 6, 8\}$. In the right figure, all the nodes on the thickened path are in the spanning tree but only those vertices drawn as hollowed dots are the internal nodes in the simplified spanning tree.

Notice that $M(u_1)$ may not be unique in which case we can pick any one that maximizes the similarity measure.

For a tree T, we define a partial order $\preceq$ on its nodes: for two nodes u, v, $u \preceq v$ if $u = v$ or u is a descendant of v. For a set R of nodes in T, if there exists a node $u \in R$ so that for any $r \in R$, $r \preceq u$ ($u \preceq r$), then we let $\max R$ ($\min R$) be u; otherwise $\max R$ ($\min R$) is not defined. Denote by $\text{lca}(u, v)$ the lowest common ancestor(LCA) of u and v, i.e. $\text{lca}(u, v) = \min\{w | u \preceq w, v \preceq w\}$. For any two nodes u, v in a tree, denote $P(u, v)$ the unique path connecting u and v and $P(u)$ the path from u to the root. The size of a path is defined to be the number of nodes on the path. Two paths are disjoint if they do not share interior nodes. For a subset $L \subset U$, the *spanning tree* of L is defined as the minimum set of edges that connect all the leaves whose labels are in $L \cap L(T)$. Clearly, if two nodes u, v are in the spanning tree of L, so is $\text{lca}(u, v)$. A spanning tree may contain degree two vertices. A maximal degree two path in a spanning tree is a maximal path between two nodes u, v for $u \preceq v$ where all the internal nodes in the path are of degree two in the spanning tree. The *simplified spanning tree* $T(L)$ of L is the rooted tree obtained from the spanning tree of S by replacing each maximal degree two path with an edge between the two ending nodes (Figure 1). $T(L)$ can be computed easily.

Lemma 1. *For any tree T with n leaves, we can preprocess it in $O(n)$ time so that for any $L \subset U$, $T(L)$ can be computed in time $O(|L| \log n)$.*

Proof: In the preprocessing, we sort all the leaves in T according to the in-order traversal. We also construct a data structure for answering lowest

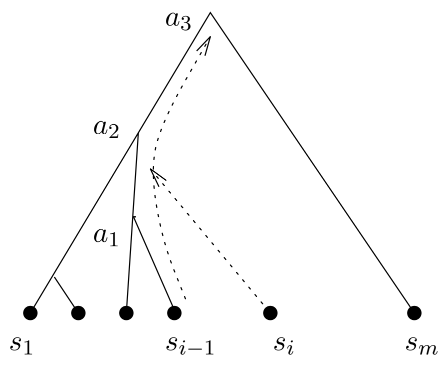

Figure 2. Search a bay.

common ancestor query in $O(1)$ time. The preprocessing can be done in $O(n)$ time [5]. For any $L \subset U$, we first find the leaves with labels in L. Suppose that they are $s_1, s_2, \cdots, s_m$ in the order, where $m = L \cap L(T)$. This can be done in $O(|L| \log n)$ time. We query the LCA data structure to compute $\mathrm{lca}(s_1, s_m)$. We then insert $s_2, \cdots, s_{m-1}$ one by one and construct $T(L)$ incrementally. When inserting s_i, we consider the path (or the bay) between s_{i-1} and s_m on the already constructed piece of $T(L)$ (Figure 2). Suppose that the nodes on the bay are $s_{i-1} = a_0, a_1, \cdots, a_k = \mathrm{lca}(s_{i-1}, s_m)$. We perform a binary search to locate the edge $a_j a_{j+1}$ so that $a_j \preceq \mathrm{lca}(s_{i-1}, s_i) \preceq a_{j+1}$. If such an edge does not exist, we insert s_i to the edge $a_k s_m$ at the node $\mathrm{lca}(s_i, s_m)$. Each insertion takes $O(\log m)$ time as each LCA query takes $O(1)$ time. Therefore, $T(L)$ can be constructed in $O(|L| \log n)$ time. □

For any node $u \in T$, let $D(u, L) = \{v | v \preceq u \text{ and } v \in T(L)\}$ be all the nodes in $T(L)$ that are under u. The following is useful.

Lemma 2. *If $D(u, L) \neq \emptyset$, then $\max D(u, L)$ is defined. Further, given $T(L)$, we can preprocess it into a data structure in $O(|L|)$ time so that $D(u, L)$ can be computed in $O(\log |L|)$ time for any $u \in T$.*

Proof: Consider the subtrees T_1, T_2 rooted at u's two children, respectively. If none of T_1, T_2 contains any node in $T(L)$, then $D(u, L) = \emptyset$. If exactly one of T_1, T_2, say T_1, contains node in $T(L)$, then $\max D(u, L)$ is the highest node in $T(L) \cap T_1$. If both of T_1, T_2 contain node in $T(L)$, then u must be in $T(L)$, i.e. $\max D(u, L) = u$.

Given $T(L)$, we can sort all the leaves in $T(L)$ according to the in-order traversal. Suppose that they are $s_1, s_2, \cdots, s_m$. For each i, we also store in an array all the nodes on the path between s_i and s_{i+1} in $T(L)$ in the order they appear on the path. For any given node u, we can first perform a binary search to find i so that u is between s_i and s_{i+1} in the in-order traversal (if such i does not exist, then either $D(u, L) = \emptyset$ or $D(u, L) = \text{lca}(s_1, s_m)$.). We then perform a binary search on the path between s_i and s_{i+1} to find $\max D(u, L)$. Both binary searches take $O(\log |L|)$ time. □

When computing the best match of a node u, we just need to consider all the nodes in the simplified spanning tree of $L(u)$ in the tree T_2 according to the following lemma.

Lemma 3. *For two trees T_1 and T_2 and a node u in T_1, the best matching node of u must be a node in the tree $T_2(L(u)))$ if $L(T_2) \cap L(u) \neq \emptyset$.*

Proof: Consider a node $v \notin T_2(L(u))$. If $D(v, L) = \emptyset$, then $S(u, v) = 0$. Otherwise, consider $w = \max D(v, L)$. By Lemma 2, w is well defined. Further $S(u, w) > S(u, v)$ because $L(w) \subset L(v)$ and $L(w) \cap L(u) = L(v) \cap L(u)$. Therefore, v cannot be the best match of u if $L(T_2) \cap L(u) \neq \emptyset$. □

By Lemma 1 and 3, we immediately have that

Theorem 4. *For any two trees T_1, T_2, with n leaves each, one can compute the best match for all the nodes in T_1 in $O(n)$ space and $O(\sum_{u \in T_1} |L(u)| \log n)$ time. The running time is $O(n^2 \log n)$ in the worst case and $O(n \log^2 n)$ when T_1 is balanced.*

Proof: We first traverse T_1, T_2 and store at each node the number of leaves under it. For any node $u \in T_1$, we compute $T_2(L(u))$. Then, we traverse $T_2(L(u))$ and compute for each node v in $T_2(L(u))$ the number of leaves under v in $T_2(L(u))$, i.e. $|L(u) \cap L(v)|$. For each $v \in T_2(L(u))$, we compute

$$S(u, v) = \frac{|L(u) \cap L(v)|}{|L(u) \cup L(v)|} = \frac{|L(u) \cap L(v)|}{|L(u)| + |L(v)| - |L(u) \cap L(v)|}\,.$$

We then simply pick the node that maximizes $S(u, v)$. By Lemma 3, it is the best match of u. By Lemma 1, the above can be done in $O(n)$ space and in $O(\sum_{u \in T_1} |L(u)| \log n)$ time. □

The main result of the paper is to show how to improve the worst case quadratic bound by combining a technique from Computational Geometry and a path decomposition of trees.

Theorem 5. *For any two trees T_1 and T_2 with n leaves each, the best match for all the nodes in T_1 can be computed in time $O((n \log n)^{1.5})$.*

In the following, we first specify a sub-problem and an algorithm for it. We then show how to use it as a subroutine to solve the matching problem.

3 Ancestor match problem

Consider the following problem: given two trees T_1, T_2 and a node $u \in T_1$, compute the best matching nodes for all the ancestors of u. We call this problem *ancestor match problem.* In this section, we describe an algorithm for this problem. In the next section, we show how to use the algorithm developed in this section to solve the best match problem.

For the ancestor match problem, we show the following result.

Theorem 6. *Given two trees T_1, T_2 and a node u in T_1, we can preprocess it into a data structure in $O(|L(u)| \log n)$ time and space so that for any ancestor w of u, $M(w)$ can be computed in time $O((|L(w)| - |L(u)|) \log^2 n)$.*

Set $L_2 = L(w) \setminus L(u)$ and $m = |L_2|$. According to Lemma 3, to be the best match of w, a node has to be in the tree $T_2(L(w))$. For a node v in $T_2(L(w))$, we distinguish three cases (Figure 3):

- Type 1. $v \in T_2(L(u))$,
- Type 2. $v \in T_2(L_2)$, and
- Type 3. $v \notin T_2(L(u))$ and $v \notin T_2(L_2)$.

Note that a node can be of both Type 1 and Type 2. Clearly, there are at most $O(m)$ Type 2 nodes. For any Type 3 node v, consider its children s_1, s_2 in $T_2(L(w))$. Because v is not in $T_2(L(u))$, one of s_1, s_2 must be in $T_2(L_2)$. Therefore, the number of Type 3 nodes is bounded by the number of Type 2 nodes and is $O(m)$ as well. In the following, we show that given $T_2(L(u))$, after $O(|L(u)|)$ time preprocessing, we can compute $S(w, v)$ in $O(m \log n)$ time for all the v's of Type 2 and 3.

We preprocess $T_2(L(u))$ as in Lemma 2. With each node in $T_2(L(u))$, we also store the number of leaves under it in $T_2(L(u))$. Then, we compute $T_2(L_2)$ and store with each node in $T_2(L_2)$ the number of leaves under it in

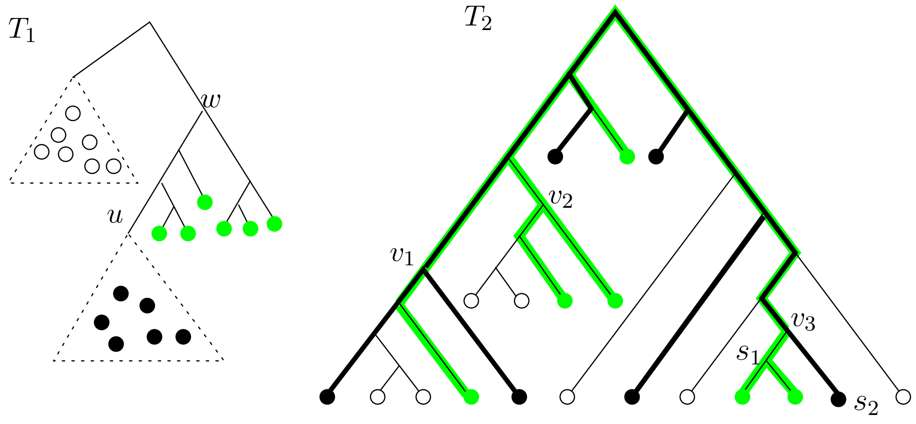

Figure 3. Computing $M(w)$. The black and shaded dots represent leaves in $L(u)$ and $L_2 = L(w) \setminus L(u)$, respectively. The hollowed dots are leaves not in $L(w)$. v_1, v_2, v_3 are examples of nodes of Type 1, 2, and 3, respectively. In the right figure, $T_2(L(u))$ is shown in thickened edges; and $T_2(L_2)$ in shaded edges. $s_1 = \max D(v_3, L(u))$ and $s_2 = \max D(v_3, L_2)$.

$T_2(L_2)$. To discover Type 3 nodes, we compute for each node v in $T_2(L_2)$, the node at which v is attached to $T_2(L(u))$. This can be done by a binary search similar to Lemma 2. For any node v in T_2, we can find $s_1 = \max D(v, L(u))$ and $s_2 = \max D(v, L_2)$ in $O(\log n)$ time after appropriate preprocessing as in Lemma 2 (Figure 3). Then, we have that $|L(w) \cap L(v)| = |L(u) \cap L(v)| + |L_2 \cap L(v)| = |L(u) \cap L(s_1)| + |L_2 \cap L(s_2)|$. Since $s_1 \in T_2(L(u))$, $|L(u) \cap L(s_1)|$ is the number of leaves under s_1 in $T_2(L(u))$. Similarly, $|L_2 \cap L(s_2)|$ is the number of leaves under s_2 in $T_2(L_2)$. Therefore, $S(w, v)$ can be computed in $O(\log n)$ time for each $v \in T_2$. Since there are $O(m)$ Type 2 and 3 nodes, we can process them in $O(m \log n)$ time.

Now, we focus on those Type 1 nodes, the nodes in $T_2(L(u))$. For a node $v \in T_2(L(u))$, we define a function $f_v : \mathbb{R}^2 \to \mathbb{R}$ where

$$f_v(x, y) = \frac{|L(u) \cap L(v)| + x}{|L(u) \cup L(v)| + y}.$$

Recall that $L_2 = L(w) \setminus L(u)$ and $m = |L_2|$. Set $\alpha(v) = L(v) \cap L_2$. We make the following observation.

$$\begin{aligned}
S(w,v) &= \tfrac{|L(w)\cap L(v)|}{|L(w)\cup L(v)|} \\
&= \tfrac{|L(u)\cap L(v)|+|L_2\cap L(v)|}{|L(u)\cup L(v)|+|L(w)\setminus(L(u)\cup L(v))|} \\
&= \tfrac{|L(u)\cap L(v)|+|L_2\cap L(v)|}{|L(u)\cup L(v)|+|L_2|-|L_2\cap L(v)|} \\
&= f_v(|L_2\cap L(v)|, m-|L_2\cap L(v)|) \\
&= f_v(\alpha(v), m-\alpha(v))
\end{aligned}$$

Consider $D(v,L_2)$, the set of $T_2(L_2)$ nodes under v. If $D(v,L_2)=\emptyset$ then $\alpha(v)=0$, and therefore $S(w,v)=f_v(0,m)$. Otherwise, suppose that $v_1=\max D(v,L_2)$. As shown before, $\alpha(v)=\alpha(v_1)$. Then, we have that $S(w,v)=f_v(\alpha(v_1),m-\alpha(v_1))$. For any $p\in D(v,L_2)$, since p is a descendant of v_1, $\alpha(p)\le\alpha(v_1)$, i.e. $f_v(\alpha(v_1),m-\alpha(v_1))\ge f_v(\alpha(p),m-\alpha(p))$. Since $f_v(0,m)\le f_v(\alpha(p),m-\alpha(p))$ for any $p\in D(v,L_2)$, we can combine the two cases into one formula:

$$S(w,v)=\max(\max_{p\in D(v,L_2)} f_v(\alpha(p),m-\alpha(p)), f_v(0,m))\,. \tag{1}$$

For any set of nodes R, we define the maximum map $F_R(x,y)$ as

$$F_R(x,y)=\max_{v\in R} f_v(x,y)\,.$$

Further, for any node $p\in T_2(L_2)$, let $G_p(x,y)=F_{P(p)\cap T_2(L_2)}(x,y)$, the maximum map of f_v's where v is in the simplified spanning tree of L_2 and is an ancestor of p. Alternatively, $G_p(x,y)=F_{\{v\in T_2(L(u))|p\in D(v,L_2)\}}(x,y)$. Let $H(x,y)=F_{T_2(L(u))}(x,y)$. From the previous formula, we can derive the following by exchanging the order of maximization:

Lemma 7.

$$\max_{v\in T_2(L(u))} S(w,v)=\max(\max_{p\in T_2(L_2)} G_p(\alpha(p),m-\alpha(p)), H(0,m))\,. \tag{2}$$

Proof:

$$\begin{aligned}
&\max_{v\in T_2(L(u))} S(w,v) \\
=\ &\max_{v\in T_2(L(u))}(\max(\max_{p\in D(v,L_2)} f_v(\alpha(p),m-\alpha(p)), f_v(0,m))) \\
=\ &\max(\max_{v\in T_2(L(u))}(\max_{p\in D(v,L_2)} f_v(\alpha(p),m-\alpha(p))), \max_{p\in T_2(L(u))} f_v(0,m))
\end{aligned}$$

Since $p \in D(v, L_2)$ is equivalent to $v \in P(p)$, we have that

$$\max_{v \in T_2(L(u))} \max_{p \in D(v,L_2)} f_v(x,y) = \max_{p \in T_2(L_2)} \max_{v \in P(p) \cap T_2(L(u))} f_v(x,y)\,.$$

Thus,

$$\begin{aligned}
&\max_{v \in T_2(L(u))} S(w,v) \\
= \;& \max(\max_{p \in T_2(L_2)} (\max_{v \in P(p) \cap T_2(L_2)} f_v(\alpha(p), m-\alpha(p))), \max_{v \in T_2(L(u))} f_v(0,m)) \\
= \;& \max(\max_{p \in T_2(L_2)} G_p(\alpha(p), m-\alpha(p)), H(0,m))
\end{aligned}$$

□

By the above lemma, the problem reduces to computing G_p, H. We now exploit the geometry of the function F_R and the combinatorial structure of trees to show that after $O(|L(u)| \log n)$ preprocessing, for any $p \in T_2$ and any x, y, we can compute $G_p(x,y)$ and $H(x,y)$ in $O(\log^2 n)$ time and therefore complete the proof of Theorem 6.

For any function $f : \mathbb{R}^2 \to \mathbb{R}$, we can alternatively consider its image $(x, y, f(x,y))$ in three dimensions. In Computational Geometry, the maximum map $F_R(x,y)$ is also called the upper envelope of the surfaces defined by f_v's for $v \in R$. The combinatorial complexity of the upper envelope of surfaces has been a well studied subject in geometry. It is well known that for n surfaces defined by constant degree algebraic or rational functions, their upper envelope can have combinatorial complexity of $\Theta(n^2)$. Luckily, the function $f_v(x,y)$ we study here satisfies special properties so that $F_R(x,y)$ has linear complexity in terms of the size of R. By definition, each function f_v has the form $\frac{a+x}{b+y}$. For any two such surfaces defined by $z = \frac{a_1+x}{b_1+y}$ and $z = \frac{a_2+x}{b_2+y}$, their intersection, when projected to the xy plane, is a straight line defined by $(b_1 - b_2)x - (a_1 - a_2)y + a_2 b_1 - a_1 b_2 = 0$. Therefore, two such surfaces intersect at a topological line in the space, and three such surfaces intersect at at most one point. Algebraic surfaces with such properties are called pseudo planes as the arrangement of such surfaces has similar structure to the arrangement of planes. In particular, the combinatorial complexity of the upper-envelope of n pseudo planes is $O(n)$ and can be computed in time $O(n \log n)$ [7]. To summarize,

Lemma 8. *For a set of functions $\{f_i | 1 \leq i \leq k\}$ where $f_i(x, y) = \frac{a_i + x}{b_i + y}$, we can preprocess it into a data structure with storage $O(k)$ in $O(k \log k)$ time so that for any (x, y), $F(x, y) = \max_i f_i(x, y)$ can be computed in time $O(\log k)$.*

By Lemma 8, we can compute $H(x, y)$ in $O(\log n)$ time after preprocessing that takes $O(|L(u)|)$ space and $O(|L(u)| \log n)$ time. But we cannot afford to compute G_p for all possible p's because that would require quadratic space and time. To reduce the preprocessing complexity, we need a tool called *canonical path representations* of $T_2(L_2)$. For any tree T with n nodes, a set $\mathcal{P}(T)$ of paths in T is a canonical path representation of T if it satisfies the following properties:

- $|\mathcal{P}| = O(n)$;
- $\sum_{P \in \mathcal{P}} |P| = O(n \log n)$; and
- any path P in T can be represented as the union of $O(\log n)$ paths in $\mathcal{P}$, which are called the canonical representation of P.

It is known that there always exists a canonical path representation for any tree. It can be achieved by decomposing the tree into paths as in Sleator and Tarjan's link-cut tree, computing a weighted balanced binary on top of each solid path where the weight of each node is the number of nodes in the subtree incident to the dashed edge, and creating a piece for each node in that tree [13, 9]. To summarize, we have that

Lemma 9. *For any tree T, $\mathcal{P}(T)$ can be computed in $O(n \log n)$ time and space. Further, for any path P in T, its canonical representation can be computed in time $O(\log n)$.*

Now, we can compute a canonical path representation $\mathcal{P}$ for $T_2(L(u))$. For each $P \in \mathcal{P}$, we construct a data structure for efficient computation of F_P as in Lemma 8. Then, for any path P' in $T_2(L(u))$, we can compute a canonical representation of P'. Suppose it is $P_1, P_2, \cdots, P_k$ where $k = O(\log |L(u)|)$. Clearly, $F_{P'}(x, y) = \max_{i=1}^{k} F_{P_i}(x, y)$. Therefore, we can compute $F_{P'}(x, y)$ in $O(\log^2 |L(u)|)$ time as $k = O(\log |L(u)|)$, and computing $F_{P_i}(x, y)$ takes $O(\log |L(u)|)$ time by the data structure in Lemma 8. As for the preprocessing time, if we construct the upper-envelope for each path in $\mathcal{P}$ separately, then it will be $\sum_{P_i \in \mathcal{P}} |P_i| \log |P_i| = O(|L(u)| \log^2 n)$. But if we use a divide and conquer method to construct the upper envelope, it is easy to show that the preprocessing only takes $O(|L(u)| \log n)$ time as the intermediate results in the divide and conquer construction correspond to the upper envelope of the pieces on a solid path in the link-cut tree. The details are omitted in this abstract.

This completes the proof of Theorem 6.

4 The $O((n \log n)^{1.5})$ algorithm

In this section, we shall show how to use the algorithm for the ancestor match problem as a subroutine to solve the best match problem.

The algorithm works recursively. For the tree T_1, we find a node u as follows. We first find a node u' in the tree so that $n/3 \leq |L(u')| \leq 2n/3$. This is always possible for a binary tree. Consider the path $P(u')$, the path from u' to the root. If the length of the path is shorter than $\delta(n)$, where $\delta(n)$ will be chosen later, we simply set u to be u'. Otherwise, we choose u to be the node on $P(u')$ which is $\delta(n)$ away from the root. Now, we build a data structure for the ancestor match problem for u according to Theorem 6. Suppose that the nodes on $P(u)$ are $u = u_1, u_2, \cdots, u_{\delta(n)}$ in the ascending order. We compute the best match for u_i's, where $2 \leq i \leq k$, by querying the precomputed data structure. We then delete the nodes $u_2, u_3, \cdots, u_{\delta(n)}$ (and the incident edges) and recursively solve the best match problem for each of $\delta(n)$ subtrees. Notice that we only need to preprocess T_2 as in Lemma 1 so that $T_2(L)$ can be computed efficiently for any L. This precomputation is done only once.

As for the space used by the algorithm, it is clearly $O(n \log n)$ as the top level recursion dominates. As for the running time, we consider the time used for preprocessing and query separately. Let $t_1(n), t_2(n)$ denote the total time spent for preprocessing and query, respectively, for trees with n leaves. Let $m_1 = |L(u)|$ and $m_i = |L(u_i)| - |L(u_{i-1})|$ for $2 \leq i \leq k$. For preprocessing, we have the recurrence:

$$t_1(n) \leq \sum_{i=1}^{k} t_1(m_i) + m_1 \log n \, ,$$

where $\sum_{i=1}^{k} m_i = n$ and $n/3 \leq m_1 \leq n - \delta(n)$. It is easy to see that $t_1(n) = O(n^2 \log n / \delta(n))$ for $\delta(n) = O(n^{c_1})$ for any constant $c_1 < 1$.

According to Theorem 6, the cost of computing the best match of u_i, where $2 \leq i \leq k$, is $O((|L(u_i)| - m_1)\log^2 n)$. We can charge the query cost $O(\log^2 n)$ to each leaf node in $L(u_i) \setminus L(u_1)$. Since the length of $P(u)$ is at most $\delta(n)$, the total charge a node may receive is $O(\delta(n)\log^2 n)$ in one level of recursion. If we denote $t_3(n)$ the maximum total charge a node may get during the entire algorithm, we then have that $t_3(n) \leq t_3(2n/3) + \delta(n)\log^2 n$ since $m_i \leq 2n/3$ for $2 \leq i \leq k$. Thus, $t_3(n) = O(\delta(n)\log^2 n)$ if $\delta(n) = \Omega(n^{c_2})$ for any constant $c_2 > 0$. Therefore , the total running time of the algorithm is $O(n^2 \log n/\delta(n) + n\delta(n)\log^2 n)$. By setting $\delta(n) = \sqrt{n/\log n}$, we obtain the running time of $O((n\log n)^{1.5})$ and thus prove Theorem 5.

5 Conclusion

In this paper, we consider the problem of computing best matching nodes between two trees according to the set similarity measure. We first show a simple algorithm that runs in quadratic time in the worst case but in roughly linear time for balanced trees. That algorithm has been implemented in the phylogenetic tree visualization project ongoing in HP(formerly Compaq) Systems Research Center and has proven to be quite efficient in practice.

Next, we present an algorithm that computes the best matching nodes between two trees in sub-quadratic time in the worst case. We achieve the bound by reducing the problem to computing the maximum map (upper-envelope) of a set of pseudo-planes and by applying the results for the upper-envelope of pseudo-planes. It is interesting to know whether our technique can be extended to solving other problems. Of course, the major question left open is whether we can solve the best match problem in roughly linear time.

Acknowledgment

The work is motivated by the phylogenetic tree visualization project at Systems Research Center. The author wishes to thank the other project team members, Valerie King, Tamara Munzner, Serdar Tasiran, and Yunhong Zhou, for their useful discussion.

References

[1] E. N. Adams. Consensus techniques and the comparison of taxonomic trees. *Systematic Zoology*, 21:390–397, 1972.

[2] A. Z. Broder. On the resemblance and containment of documents. In *SEQS: Sequences*, pages 21–29, 1998.

[3] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the Web. In *Proceedings of the Sixth International World Wide Web Conference*, pages 391–404, 1997.

[4] W. H. E. Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of Classification*, 2:7–28, 1985.

[5] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13:338–355, 1984.

[6] T. Margush and F. R. McMorris. Consensus n-trees. *Bulletin of Mathematical Biology*, 3:239–244, 1981.

[7] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, 1995.

[8] N. Shivakumar and H. García-Molina. SCAM: A copy detection mechanism for digital documents. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, 1995.

[9] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[10] R. R. Sokal and F. J. Rohlf. Taxonomic congruence in the Leptopodomorpha re-examined. *Systematic Zoology*, 30:309–325, 1981.

[11] R. Stinebrickner. s-Consensus trees and indices. *Bulletin of Mathematical Biology*, 46:923–935, 1984.

[12] R. Stinebrickner. s-Consensu index method: an additional axiom. *Journal of Classification*, 3:319–327, 1986.

[13] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[14] E. O. Wiley, D. Siegel-Causey, D. r. Brooks, and V. A. Funk. *The Compleat Cladist.* Museum of Natural History, The University of Kansas, 1991.