An Optimal Decomposition Algorithm for Tree Edit Distance

ERIK D. DEMAINE

Massachusetts Institute of Technology

SHAY MOZES

Brown University

AND

BENJAMIN ROSSMAN AND OREN WEIMANN

Massachusetts Institute of Technology

Abstract. The *edit distance* between two ordered rooted trees with vertex labels is the minimum cost of transforming one tree into the other by a sequence of elementary operations consisting of deleting and relabeling existing nodes, as well as inserting new nodes. In this article, we present a worst-case $O(n^3)$-time algorithm for the problem when the two trees have size n, improving the previous best $O(n^3 \log n)$-time algorithm. Our result requires a novel adaptive strategy for deciding how a dynamic program divides into subproblems, together with a deeper understanding of the previous algorithms for the problem. We prove the optimality of our algorithm among the family of *decomposition strategy* algorithms—which also includes the previous fastest algorithms—by tightening the known lower bound of $\Omega(n^2 \log^2 n)$ to $\Omega(n^3)$, matching our algorithm's running time. Furthermore, we obtain matching upper and lower bounds for decomposition strategy algorithms of $\Theta(nm^2(1 + \log \frac{n}{m}))$ when the two trees have sizes m and n and $m < n$.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures; pattern matching*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*

General Terms: Algorithms, Theory

A preliminary version of this article [Demaine et al. 2007] appeared in the *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*.

O. Weimann is now affiliated with Weizmann Institute of Science.

Authors' addresses: E. D. Demaine, MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139; S. Mozes, Computer Science, Brown University, Providence, RI 02912-1910; B. Rossman, O. Weimann (corresponding author), Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel, e-mail: Oren.weimann@weizmann.ac.il.

DOI 10.1145/1644015.1644017 http://doi.acm.org/10.1145/1644015.1644017

Additional Key Words and Phrases: Decomposition strategy, dynamic programming, edit distance, ordered trees, tree edit distance

ACM Reference Format:
Demaine, E. D., Mozes, S., Rossman, B., and Weimann, O. 2009. An optimal decomposition algorithm for tree edit distance. ACM Trans. Algor. 6, 1, Article 2 (December 2009), 19 pages.
DOI = 10.1145/1644015.1644017 http://doi.acm.org/10.1145/1644015.1644017

1. *Introduction*

The problem of comparing trees occurs in diverse areas such as structured text databases like XML, computer vision, compiler optimization, natural language processing, and computational biology [Bille 2005; Chawathe 1999; Klein et al. 2000; Shasha and Zhang 1989; Tai 1979].

One major application is the analysis of RNA molecules in computational biology. *Ribonucleic acid* (RNA) is a polymer consisting of a sequence of nucleotides (Adenine, Cytosine, Guanine, and Uracil) connected linearly via a backbone. In addition, complementary nucleotides (AU, GC, and GU) can form hydrogen bonds, leading to a structural formation called the *secondary structure* of the RNA. Because of the nested nature of these hydrogen bonds, the secondary structure of RNA can be naturally represented by an ordered rooted tree [Gusfield 1997; Waterman 1995] as depicted in Figure 1. Recently, comparing RNA sequences has gained increasing interest thanks to numerous discoveries of biological functions associated with RNA. A major fraction of RNA's function is determined by its secondary structure [Moore 1999]. Therefore, computing the similarity between the secondary structure of two RNA molecules can help determine the functional similarities of these molecules.

The *tree edit distance* metric is a common similarity measure for rooted ordered trees. It was introduced by Tai in the late 1970's [Tai 1979] as a generalization of the well-known string edit distance problem [Wagner and Fischer 1974]. Let F and G be two rooted trees with a left-to-right order among siblings and where each vertex is assigned a label from an alphabet Σ. The *edit distance* between F and G is the minimum cost of transforming F into G by a sequence of elementary operations consisting of deleting and relabeling existing nodes, as well as inserting new nodes (allowing at most one operation to be performed on each node). These operations are illustrated in Figure 2. Formally, given a node v in F with parent v', *relabel* changes the label of v, *delete* removes a nonroot node v and sets the children of v as the children of v' (the children are inserted in the place of v as a subsequence in the left-to-right order of the children of v'), and *insert* (the complement of delete) connects a new node v as a child of some v' in F making v the parent of a consecutive subsequence of the children of v'. The cost of the elementary operations is given by two functions, c_{del} and c_{match}, where $c_{\text{del}}(\tau)$ is the cost of deleting or inserting a vertex with label τ, and $c_{\text{match}}(\tau_1, \tau_2)$ is the cost of changing the label of a vertex from τ_1 to τ_2. Since a deletion in F is equivalent to an insertion in G and vice versa, we can focus on finding the minimum cost of a sequence of just deletions and relabelings in both trees that transform F and G into isomorphic trees.

1.1. Previous Results.

To state running times, we need some basic notation. Let n and m denote the sizes $|F|$ and $|G|$ of the two input trees, ordered so that $n \geq m$. Let n_{leaves} and m_{leaves} denote the corresponding number of leaves in each

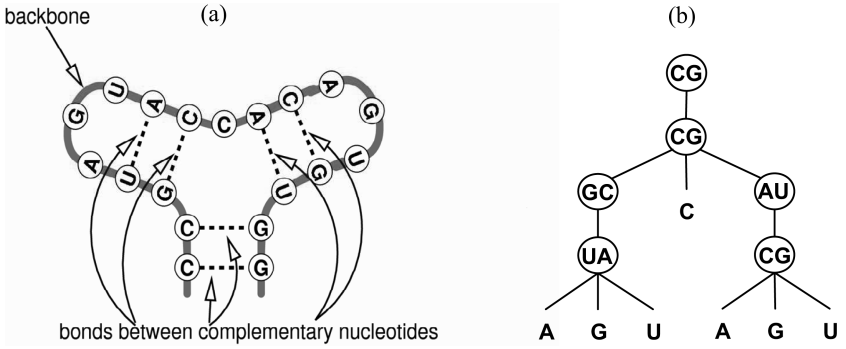

FIG. 1. Two different ways of viewing an RNA sequence. In (a), a schematic 2-dimensional description of an RNA folding. In (b), the RNA as a rooted ordered tree.

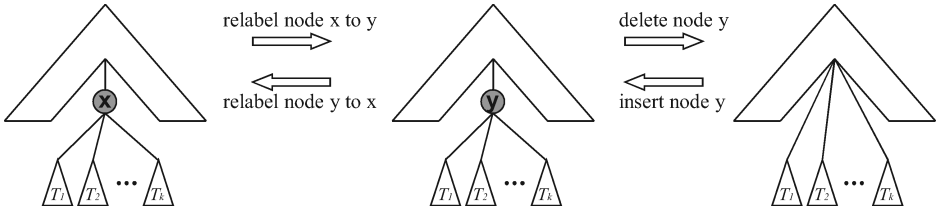

FIG. 2. The three editing operations on a tree with vertex labels.

tree, and let n_{height} and m_{height} denote the corresponding height of each tree, which can be as large as n and m, respectively.

Tai [1979] presented the first algorithm for computing tree edit distance, which requires $O(n^2_{\text{leaves}}m^2_{\text{leaves}}nm)$ time and space, and thus has a worst-case running time of $O(n^3m^3) = O(n^6)$. Shasha and Zhang [1989] improved this result to an $O(\min\{n_{\text{height}}, n_{\text{leaves}}\}\cdot\min\{m_{\text{height}}, m_{\text{leaves}}\}\cdot nm)$ time algorithm using $O(nm)$ space. In the worst case, their algorithm runs in $O(n^2m^2) = O(n^4)$ time. Klein [1998] improved this result to a worst-case $O(m^2n \log n) = O(n^3 \log n)$ time algorithm using $O(nm)$ space. These last two algorithms are based on closely related dynamic programs, and both present different ways of computing only a subset of a larger dynamic program table; these entries are referred to as *relevant subproblems*. Dulucq and Touzet [2003] introduced the notion of a *decomposition strategy* (see Section 2.3) as a general framework for algorithms that use this type of dynamic program, and proved a lower bound of $\Omega(nm \log n \log m)$ time for any such strategy.

Many other solutions have been developed; see Apostolico and Galil [1997], Bille [2005], and Valiente [2002] for surveys. The most recent development is by Chen [2001], who presented a different approach that uses results on fast matrix multiplication. Chen's algorithm uses $O(nm + nm^2_{\text{leaves}} + n_{\text{leaves}}m^{2.5}_{\text{leaves}})$ time and $O(n + (m + n^2_{\text{leaves}})\min\{n_{\text{leaves}}, n_{\text{height}}\})$ space. In the worst case, this algorithm runs in $O(nm^{2.5}) = O(n^{3.5})$ time. Among all these algorithms, Klein's is the fastest in terms of worst-case time complexity, and previous improvements to Klein's $O(n^3 \log n)$ time bound were achieved only by constraining the edit operations or the scoring scheme [Chawathe 1999; Selkow 1977; Shasha and Zhang 1990; Zhang 1995].

1.2. OUR RESULTS. We present a new algorithm for computing the tree edit distance that falls into the same *decomposition strategy* framework of Dulucq and Touzet [2003], Klein [1998], and Shasha and Zhang [1989]. In the worst case, our algorithm requires $O(nm^2(1 + \log \frac{n}{m})) = O(n^3)$ time and $O(nm)$ space. The corresponding sequence of edit operations can easily be obtained within the same time and space bounds. We therefore improve upon all known algorithms in the worst-case time complexity. Our approach is based on Klein's, but whereas the recursion scheme in Klein's algorithm is determined by just one of the two input trees, in our algorithm the recursion depends alternately on both trees. Furthermore, we prove a worst-case lower bound of $\Omega(nm^2(1 + \log \frac{n}{m}))$ time for all decomposition strategy algorithms. This bound improves the previous best lower bound of $\Omega(nm \log n \log m)$ time [Dulucq and Touzet 2003], and establishes the optimality of our algorithm among all decomposition strategy algorithms.

1.3. ROADMAP. In Section 2 we give a simple and unified presentation of the two well-known tree edit algorithms, on which our algorithm is based, and on the class of decomposition strategy algorithms. We present and analyze the time complexity of our algorithm in Section 3, and prove the matching lower bound in Section 4. An explicit $O(nm)$ space complexity version of our algorithm is given in Section 5 and final conclusions are presented in Section 6.

2. *Background and Framework*

Both the existing algorithms and ours compute the edit distance of finite ordered Σ-labeled forests, henceforth *forests*. These are forests that have a left-to-right order among siblings and each vertex is assigned a label from a given finite alphabet Σ such that two different vertices can have the same label or different labels. The unique empty forest/tree is denoted by $\emptyset$. The vertex set of a forest F is written simply as F, as when we speak of a vertex $v \in F$. For a forest F and $v \in F$, $\sigma(v)$ denotes the label of v, F_v denotes the subtree of F rooted at v, and $F - v$ denotes the forest F after deleting v. The special case of $F - root(F)$ where F is a tree and $root(F)$ is its root is denoted F°. The leftmost and rightmost trees of a forest F are denoted by L_F and R_F and their roots by ℓ_F and r_F. We denote by $F - L_F$ the forest F after deleting the entire leftmost tree L_F; similarly $F - R_F$. A left-to-right postorder traversal of F is the postorder traversal of all its trees $L_F, \ldots, R_F$ from left to right. For a tree T, the postorder traversal is defined recursively as the postorder traversal of the forest T° followed by a visit of $root(T)$ (as opposed to a preorder traversal that first visits $root(T)$ and then T°). A forest obtained from F by a sequence of any number of deletions of the leftmost and rightmost roots is called a *subforest* of F.

Given forests F and G and vertices $v \in F$ and $w \in G$, we write $c_{\text{del}}(v)$ instead of $c_{\text{del}}(\sigma(v))$ for the cost of deleting or inserting $\sigma(v)$, and we write $c_{\text{match}}(v, w)$ instead of $c_{\text{match}}(\sigma(v), \sigma(w))$ for the cost of relabeling $\sigma(v)$ to $\sigma(w)$. $\delta(F, G)$ denotes the edit distance between the forests F and G.

Because insertion and deletion costs are the same (for a node of a given label), insertion in one forest is tantamount to deletion in the other forest. Therefore, the only edit operations we need to consider are relabelings and deletions of nodes in both forests. In the next two sections, we briefly present the algorithms of Shasha and Zhang [1989] and of Klein [1998]. This presentation, inspired by the tree similarity

survey of Bille [2005], is somewhat different from the original presentations and is essential for understanding our algorithm.

2.1. SHASHA AND ZHANG'S ALGORITHM. Given two forests F and G of sizes n and m, respectively, the following lemma is easy to verify. Intuitively, the lemma says that in any sequence of edit operations the two rightmost roots in F and G must either be matched with each other or else one of them is deleted.

LEMMA 2.1. [SHASHA AND ZHANG 1989]. *$\delta(F, G)$ can be computed as follows:*

—$\delta(\emptyset, \emptyset) = 0$;
—$\delta(F, \emptyset) = \delta(F - r_F, \emptyset) + c_{\text{del}}(r_F)$;
—$\delta(\emptyset, G) = \delta(\emptyset, G - r_G) + c_{\text{del}}(r_G)$;
—$$\delta(F, G) = \min \begin{cases} \delta(F - r_F, G) + c_{\text{del}}(r_F), \\ \delta(F, G - r_G) + c_{\text{del}}(r_G), \\ \delta(R_F^\circ, R_G^\circ) + \delta(F - R_F, G - R_G) + c_{\text{match}}(r_F, r_G). \end{cases}$$

Lemma 2.1 yields an $O(m^2n^2)$ dynamic programming algorithm. If we index the vertices of the forests F and G according to their left-to-right postorder traversal position, then entries in the dynamic program table correspond to pairs (F', G') of subforests F' of F and G' of G where F' contains vertices $\{i_1, i_1 + 1, \ldots, j_1\}$ and G' contains vertices $\{i_2, i_2 + 1, \ldots, j_2\}$ for some $1 \le i_1 \le j_1 \le n$ and $1 \le i_2 \le j_2 \le m$.

However, we next show that only $O(\min\{n_{\text{height}}, n_{\text{leaves}}\} \cdot \min\{m_{\text{height}}, m_{\text{leaves}}\} \cdot nm)$ different *relevant subproblems* are encountered by the recursion computing $\delta(F, G)$. We calculate the number of *relevant subforests* of F and G independently, where a forest F' (respectively, G') is a relevant subforest of F (respectively, G) if it occurs in the computation of $\delta(F, G)$. Clearly, multiplying the number of relevant subforests of F and of G is an upper bound on the total number of relevant subproblems.

We now count the number of relevant subforests of F; the count for G is similar. First, notice that for every node $v \in F$, F_v° is a relevant subproblem. This is because the recursion allows us to delete the rightmost root of F repeatedly until v becomes the rightmost root; we then match v (i.e., relabel it) and get the desired relevant subforest. A more general claim is stated and proved later on in Lemma 2.3. We define

$$\text{keyroots}(F) = \{ \text{the root of } F\} \cup \{v \in F \mid v \text{ has a left sibling}\}.$$

It is easy to see that every relevant subforest of F is a prefix (with respect to the postorder indices) of F_v° for some node $v \in \text{keyroots}(F)$. If we define v's collapse depth $\text{cdepth}(v)$ to be the number of keyroot ancestors of v, and $\text{cdepth}(F)$ to be the maximum $\text{cdepth}(v)$ over all nodes $v \in F$, we get that the total number of relevant subforest of F is at most

$$\sum_{v \in \text{keyroots}(F)} |F_v| = \sum_{v \in F} \text{cdepth}(v) \le \sum_{v \in F} \text{cdepth}(F) = |F| \text{cdepth}(F).$$

This means that given two trees, F and G, of sizes n and m we can compute $\delta(F, G)$ in $O(\text{cdepth}(F) \cdot \text{cdepth}(G) \cdot nm) = O(n_{\text{height}} \cdot m_{\text{height}} \cdot nm)$ time. Shasha and Zhang also proved that for any tree T of size n, $\text{cdepth}(T) \le \min\{n_{\text{height}}, n_{\text{leaves}}\}$, hence the result. In the worst case, this algorithm runs in $O(m^2n^2) = O(n^4)$ time.

2.2. KLEIN'S ALGORITHM. Klein's [1998] algorithm is based on a recursion similar to Lemma 2.1. Again, we consider forests F and G of sizes $|F| = n \geq |G| = m$. Now, however, instead of recursing always on the rightmost roots of F and G, we recurse on the leftmost roots if $|L_F| \leq |R_F|$ and on the rightmost roots otherwise. In other words, the "direction" of the recursion is determined by the (initially) larger of the two forests. We assume the number of relevant subforests of G is $O(m^2)$; we have already established that this is an upper bound.

We next show that Klein's algorithm yields only $O(n \log n)$ relevant subforests of F. The analysis is based on a technique called *heavy path decomposition* [Harel and Tarjan 1984; Sleator and Tarjan 1983]. We mark the root of F as *light*. For each internal node $v \in F$, we pick one of v's children with maximal number of descendants and mark it as *heavy*, and we mark all the other children of v as *light*. We define $\text{ldepth}(v)$ to be the number of light nodes that are proper ancestors of v in F, and $\text{light}(F)$ as the set of all light nodes in F. It is easy to see that for any forest F and vertex $v \in F$, $\text{ldepth}(v) \leq \log |F| + O(1)$. Note that every relevant subforest of F is obtained by some $i \leq |F_v|$ consecutive deletions from F_v for some light node v. Therefore, the total number of relevant subforests of F is at most

$$\sum_{v \in \text{light}(F)} |F_v| \leq \sum_{v \in F} 1 + \text{ldepth}(v) \leq \sum_{v \in F} (\log |F| + O(1)) = O(|F| \log |F|).$$

Thus, we get an $O(m^2 n \log n) = O(n^3 \log n)$ algorithm for computing $\delta(F, G)$.

2.3. THE DECOMPOSITION STRATEGY FRAMEWORK . Both Klein's [1998] and Shasha and Zhang's [1989] algorithms are based on Lemma 2.1. The difference between them lies in the choice of when to recurse on the rightmost roots and when on the leftmost roots. The family of *decomposition strategy* algorithms based on this lemma was formalized by Dulucq and Touzet [2003].

Definition 2.2. Let F and G be two forests. A strategy is a mapping from pairs (F', G') of subforests of F and G to $\{\mathsf{left}, \mathsf{right}\}$. A decomposition algorithm is an algorithm based on Lemma 2.1 with the directions chosen according to a specific strategy.

Each strategy is associated with a specific set of recursive calls (or a dynamic programming algorithm). The strategy of Shasha and Zhang's algorithm is $S(F', G') = \mathsf{right}$ for all F', G'. The strategy of Klein's algorithm is $S(F', G') = \mathsf{left}$ if $|L_{F'}| \leq |R_{F'}|$, and $S(F', G') = \mathsf{right}$ otherwise. Notice that Shasha and Zhang's strategy does not depend on the input trees, while Klein's strategy depends only on the larger input tree. Dulucq and Touzet [2003] proved a lower bound of $\Omega(mn \log m \log n)$ time for any decomposition strategy algorithm.

The following lemma states that every decomposition algorithm computes the edit distance between every two root-deleted subtrees of F and G.

LEMMA 2.3. *Given a decomposition algorithm with strategy S, the pair (F_v°, G_w°) is a relevant subproblem for all $v \in F$ and $w \in G$ regardless of the strategy S.*

PROOF. First note that a node $v' \in F_v$ (respectively, $w' \in G_w$) is never deleted or matched before v (respectively, w) is deleted or matched. Consider the following specific sequence of recursive calls:

—Delete from F until v is either the leftmost or the rightmost root.

—Next, delete from G until w is either the leftmost or the rightmost root.

Let (F', G') denote the resulting subproblem. There are four cases to consider.

(1) *v and w are the rightmost* (*leftmost*) *roots of F' and G', and* $S(F', G') = \mathsf{right}$ (left).

Match v and w to get the desired subproblem.

(2) *v and w are the rightmost* (*leftmost*) *roots of F' and G', and* $S(F', G') = \mathsf{left}$ (right).

Note that at least one of F', G' is not a tree (since otherwise this is case (1)). Delete from one which is not a tree. After a finite number of such deletions we have reduced to case (1), either because S changes direction, or because both forests become trees whose roots are v, w.

(3) *v is the rightmost root of F', w is the leftmost root of G'.*

If $S(F', G') = \mathsf{left}$, delete from F'; otherwise delete from G'. After a finite number of such deletions this reduces to one of the previous cases when one of the forests becomes a tree.

(4) *v is the leftmost root of F', w is the rightmost root of G'.*

This case is symmetric to (3). □

3. *The Algorithm*

In this section we present our algorithm for computing $\delta(F, G)$ given two trees F and G of sizes $|F| = n \geq |G| = m$. The algorithm recursively uses a decomposition strategy in a divide-and-conquer manner to achieve $O(nm^2(1 + \log \frac{n}{m})) = O(n^3)$ running time in the worst case. For clarity we describe the algorithm recursively and analyze its time complexity. In Section 5 we prove that the space complexity of a bottom-up nonrecursive implementation of the algorithm is $O(mn) = O(n^2)$.

Before presenting our algorithm, let us try to develop some intuition. We begin with the observation that Klein's strategy always determines the direction of the recursion according to the F-subforest, even in subproblems where the F-subforest is smaller than the G-subforest. However, it is not straightforward to change this since even if at some stage we decide to choose the direction according to the other forest, we must still make sure that all subproblems previously encountered are entirely solved. At first glance this seems like a real obstacle since apparently we only add new subproblems to those that are already computed. Our key observation is that there are certain subproblems for which it is worthwhile to choose the direction according to the *currently* larger forest, while for other subproblems we had better keep choosing the direction according to the *originally* larger forest.

The *heavy path* of a tree F is the unique path starting from the root (which is light) along heavy nodes. Consider two trees, F and G, and assume we are given the distances $\delta(F_v^\circ, G_w^\circ)$ for all $v \in F$ and $w \in G$. By Lemma 2.3, these are relevant subproblems for any decomposition strategy algorithm. How would we go about computing $\delta(F, G)$ in this case? Using Shasha and Zhang's strategy would require

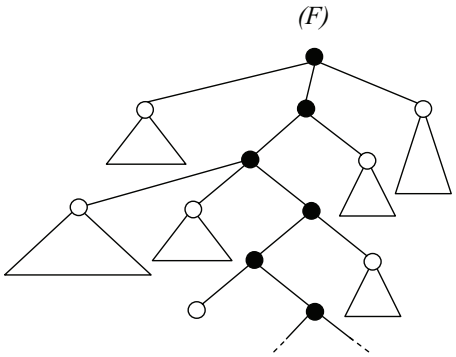

FIG. 3. A tree F with n nodes. The black nodes belong to the heavy path. The white nodes are in TopLight(F), and the size of each subtree rooted at a white node is at most $\frac{n}{2}$. Note that the root of the tree belongs to the heavy path even though it is light.

$O(|F||G|)$ time, while using Klein's strategy would take $O(|F||G|^2)$ time. Let us focus on Klein's strategy since Shasha and Zhang's strategy is independent of the trees. Note that even if we were not given the distance $\delta(F_u^\circ, G_w^\circ)$ for a node u on the heavy path of F, we would still be able to solve the problem in $O(|F||G|^2)$ time. To see why, note that in order to compute the relevant subproblem $\delta(F_u, G_w)$, we must compute all the subproblems required for solving $\delta(F_u^\circ, G_w^\circ)$ even if $\delta(F_u^\circ, G_w^\circ)$ is given.

We define the set TopLight(F) to be the set of roots of the forest obtained by removing the heavy path of F. Note that TopLight(F) is the set of light nodes with ldepth 1 in F (see the definition of ldepth in Section 2.2). This definition is illustrated in Figure 3. It follows from Lemma 2.3 that if we compute $\delta(F_v, G)$ for all $v \in$ TopLight(F), we would also compute all the subproblems $\delta(F_{v'}^\circ, G_w^\circ)$ for any $w \in G$ and v' not on the heavy path of F. Note that Klein's strategy solves $\delta(F_v, G)$ by determining the direction according to F_v even if $|F_v| < |G|$. We observe that we can do better if in such cases we determine the direction according to G. It is important to understand that making the decisions according to the larger forest when solving $\delta(F_v^\circ, G_w^\circ)$ for any $v \in F$ and $w \in G$ (i.e., regardless of whether v is on the heavy path or not) would actually increase the running time. The identification of the set TopLight(F) is crucial for obtaining the improvement.

Given these definitions, the recursive formulation of our algorithm is simply as follows.

3.1. THE ALGORITHM. We compute $\delta(F, G)$ recursively as follows:

(1) *If $|F| < |G|$, compute $\delta(G, F)$ instead.*
(2) *Recursively compute $\delta(F_v, G)$ for all $v \in$* TopLight(F).
(3) *Compute $\delta(F, G)$ using the following decomposition strategy: $S(F', G') =$* left *if F' is a tree, or if $\ell_{F'}$ is not the heavy child of its parent. Otherwise, $S(F', G') =$* right. *However, do not recurse into subproblems that were previously computed in step (2).*

The algorithm's first step makes sure that F is the larger forest, and the second step makes sure that $\delta(F_{v'}^\circ, G_w^\circ)$ is computed and stored for all v' not in the heavy path of

F and for all $w \in G$. Note that the strategy in the third step is equivalent to Klein's strategy for binary trees. For higher valence trees, this variant first makes all left deletions and then all right deletions, while Klein's strategy might change direction many times. They are equivalent in the important sense that both delete the heavy child last. The algorithm is evidentally a decomposition strategy algorithm, since for all subproblems, it either deletes or matches the leftmost or rightmost roots. The correctness of the algorithm follows from the correctness of decomposition strategy algorithms in general.

3.2. TIME COMPLEXITY. We show that our algorithm has a worst-case running time of $O(m^2 n(1 + \log \frac{n}{m})) = O(n^3)$. We proceed by counting the number of subproblems computed in each step of the algorithm. We call a subproblem trivial if at least one of the forests in this subproblem is empty. Obviously, the number of distinct trivial subproblems is $O(n^2)$. Let $R(F, G)$ denote the number of nontrivial relevant subproblems encountered by the algorithm in the course of computing $\delta(F, G)$. From now on we only count nontrivial subproblems, unless explicitly indicated otherwise.

We observe that any tree F has the following two properties:

$(*)$ $\sum_{v \in \text{TopLight}(F)} |F_v| \leq |F|$. Because F_v and $F_{v'}$ are disjoint for all $v, v' \in \text{TopLight}(F)$.

$(**)$ $|F_v| < \frac{|F|}{2}$ for every $v \in \text{TopLight}(F)$. Otherwise v would be a heavy node.

In step (2) we compute $\delta(F_v, G)$ for all $v \in \text{TopLight}(F)$. Hence, the number of subproblems encountered in this step is $\sum_{v \in \text{TopLight}(F)} R(F_v, G)$. For step (3), we bound the number of relevant subproblems by multiplying the number of relevant subforests in F and in G. For G, we count all possible $O(|G|^2)$ subforests obtained by left and right deletions. Note that for any node v' not in the heavy path of F, the subproblem obtained by matching v' with any node w in G was already computed in step (2). This is because any such v' is contained in F_v for some $v \in \text{TopLight}(F)$, so $\delta(F_{v'}^\circ, G_w^\circ)$ is computed in the course of computing $\delta(F_v, G)$ (by Lemma 2.3). Furthermore, note that in step (3), a node v on the heavy path of F cannot be matched or deleted until the remaining subforest of F is precisely the tree F_v. At this point, both matching v or deleting v result in the same new relevant subforest F_v°. This means that we do not have to consider matchings of nodes when counting the number of relevant subproblems in step (3). It suffices to consider only the $|F|$ subforests obtained by deletions according to our strategy. Thus, the total number of new subproblems encountered in step (3) is bounded by $|G|^2|F|$.

We have established that if $|F| \geq |G|$ then

$$R(F, G) \leq |G|^2|F| + \sum_{v \in \text{TopLight}(F)} R(F_v, G)$$

and if $|F| < |G|$ then

$$R(F, G) \leq |F|^2|G| + \sum_{w \in \text{TopLight}(G)} R(F, G_w).$$

We first show, by a crude estimate, that this leads to an $O(n^3)$ running time. Later, we analyze the dependency on m and n accurately.

LEMMA 3.1. $R(F, G) \leq 4(|F||G|)^{3/2}$.

PROOF. We proceed by induction on $|F|+|G|$. In the base case, $|F|+|G|=0$, so both forests are empty and $R(F,G)=0$. For the inductive step there are two symmetric cases. If $|F| \geq |G|$ then $R(F,G) \leq |G|^2|F| + \sum_{v \in \text{TopLight}(F)} R(F_v, G)$. Hence, by the induction hypothesis,

$$\begin{aligned} R(F,G) &\leq |G|^2|F| + \sum_{v \in \text{TopLight}(F)} 4(|F_v||G|)^{3/2} \\ &= |G|^2|F| + 4|G|^{3/2} \sum_{v \in \text{TopLight}(F)} |F_v|^{3/2} \\ &\leq |G|^2|F| + 4|G|^{3/2} \sum_{v \in \text{TopLight}(F)} |F_v| \ \max_{v \in \text{TopLight}(F)} \sqrt{|F_v|} \\ &\leq |G|^2|F| + 4|G|^{3/2}|F| \sqrt{\frac{|F|}{2}} = |G|^2|F| + \sqrt{8}(|F||G|)^{3/2} \leq 4(|F||G|)^{3/2}. \end{aligned}$$

Here we have used facts $(*)$ and $(**)$ and the fact that $|F| \geq |G|$. The case where $|F| < |G|$ is symmetric. □

This crude estimate gives a worst-case running time of $O(n^3)$. We now analyze the dependence on m and n more accurately. Along the recursion defining the algorithm, we view step (2) as only making recursive calls, but not producing any relevant subproblems. Rather, every new relevant subproblem is created in step (3) for a unique recursive call of the algorithm. So when we count relevant subproblems, we sum the number of new relevant subproblems encountered in step (3) over all recursive calls to the algorithm. We define sets $A, B \subseteq F$ as follows:

$$\begin{aligned} A &= \{a \in \text{light}(F) : |F_a| \geq m\} \\ B &= \{b \in F - A : b \in \text{TopLight}(F_a) \text{ for some } a \in A\} \end{aligned}$$

Note that the root of F belongs to A. Intuitively, the nodes in both A and B are exactly those for which recursive calls are made with the entire G tree. The nodes in B are the last ones, along the recursion, for which such recursive calls are made. We count separately:

(i) the relevant subproblems created in just step (3) of recursive calls $\delta(F_a, G)$ for all $a \in A$, and

(ii) the relevant subproblems encountered in the entire computation of $\delta(F_b, G)$ for all $b \in B$ (i.e., $\sum_{b \in B} R(F_b, G)$).

Together, this counts all relevant subproblems for the original $\delta(F,G)$. To see this, consider the original call $\delta(F,G)$. Certainly, the root of F is in A. So all subproblems generated in step (3) of $\delta(F,G)$ are counted in (i). Now consider the recursive calls made in step (2) of $\delta(F,G)$. These are precisely $\delta(F_v, G)$ for $v \in \text{TopLight}(F)$. For each $v \in \text{TopLight}(F)$, notice that v is either in A or in B; it is in A if $|F_v| \geq m$, and in B otherwise. If v is in B, then all subproblems arising in the entire computation of $\delta(F_v, G)$ are counted in (ii). On the other hand, if v is in A, then we are in analogous situation with respect to $\delta(F_v, G)$ as we were in when we considered $\delta(F,G)$ (i.e., we count separately the subproblems created in step (3) of $\delta(F_v, G)$ and the subproblems coming from $\delta(F_u, G)$ for $u \in \text{TopLight}(F_v)$).

Earlier in this section, we saw that the number of subproblems created in step (3) of $\delta(F,G)$ is $|G|^2|F|$. In fact, for any $a \in A$, by the same argument, the number of

subproblems created in step (3) of $\delta(F_a, G)$ is $|G|^2|F_a|$. Therefore, the total number of relevant subproblems of type (i) is $|G|^2 \sum_{a\in A} |F_a|$. For $v \in F$, define $\text{depth}_A(v)$ to be the number of proper ancestors of v that lie in the set A. We claim that $\text{depth}_A(v) \leq 1 + \log \frac{n}{m}$ for all $v \in F$. To see this, consider any sequence $a_0, \ldots, a_k$ in A where a_i is a descendent of a_{i-1} for all $i \in [1, k]$. Note that $|F_{a_i}| \leq \frac{1}{2}|F_{a_{i-1}}|$ for all $i \in [1, k]$ since the a_is are light nodes. Also note that $F_{a_0} \leq n$ and that $|F_{a_k}| \geq m$ by the definition of A. It follows that $k \leq \log \frac{n}{m}$, that is, A contains no sequence of descendants of length $> 1 + \log \frac{n}{m}$. So clearly every $v \in F$ has $\text{depth}_A(v) \leq 1 + \log \frac{n}{m}$.

We now have the number of relevant subproblems of type (i) as

$$|G|^2 \sum_{a\in A} |F_a| \leq m^2 \sum_{v\in F} 1 + \text{depth}_A(v) \leq m^2 \sum_{v\in F} \left(2 + \log \frac{n}{m}\right) = m^2 n \left(2 + \log \frac{n}{m}\right).$$

The relevant subproblems of type (ii) are counted by $\sum_{b\in B} R(F_b, G)$. Using Lemma 3.1, we have

$$\begin{aligned}\sum_{b\in B} R(F_b, G) &\leq 4|G|^{3/2} \sum_{b\in B} |F_b|^{3/2} \\ &\leq 4|G|^{3/2} \sum_{b\in B} |F_b| \max_{b\in B} \sqrt{|F_b|} \\ &\leq 4|G|^{3/2}|F|\sqrt{m} = 4m^2 n.\end{aligned}$$

Here we have used the facts that $|F_b| < m$ and $\sum_{b\in B} |F_b| \leq |F|$ (since the trees F_b are disjoint for different $b \in B$). Therefore, the total number of relevant subproblems for $\delta(F, G)$ is at most $m^2 n(2 + \log \frac{n}{m}) + 4m^2 n = O(m^2 n(1 + \log \frac{n}{m}))$. This implies the following.

THEOREM 3.2. *The running time of the algorithm is* $O(m^2 n(1 + \log \frac{n}{m}))$. □

4. *A Tight Lower Bound for Decomposition Algorithms*

In this section we present a lower bound on the worst-case running time of decomposition strategy algorithms. We first give a simple proof of an $\Omega(m^2 n)$ lower bound. In the case where $m = \Theta(n)$, this gives a lower bound of $\Omega(n^3)$ which shows that our algorithm is worst-case optimal among all decomposition algorithms. To prove that our algorithm is worst-case optimal for any $m \leq n$, we analyze a more complicated scenario that gives a lower bound of $\Omega(m^2 n(1 + \log \frac{n}{m}))$, matching the running time of our algorithm, and improving the previous best lower bound of $\Omega(nm \log n \log m)$ time [Dulucq and Touzet 2003].

In analyzing strategies we will use the notion of a *computational path*, which corresponds to a specific sequence of recursion calls. Recall that for all subforest-pairs (F', G'), the strategy S determines a direction: either right or left. The recursion can either delete from F' or from G' or match. A computational path is the sequence of operations taken according to the strategy in a specific sequence of recursive calls. For convenience, we sometimes describe a computational path by the sequence of subproblems it induces, and sometimes by the actual sequence of operations: either "delete from the F-subforest", "delete from the G-subforest", or "match".

We now turn to the $\Omega(m^2 n)$ lower bound on the number of relevant subproblems for any strategy.

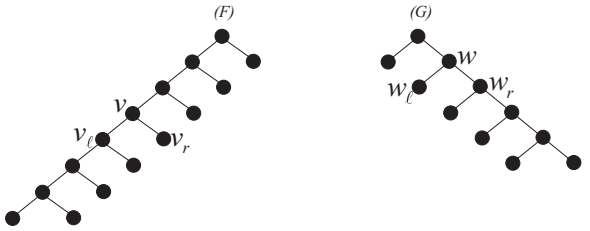

FIG. 4. The two trees used to prove an $\Omega(m^2n)$ lower bound (Lemma 4.1).

LEMMA 4.1. *For any decomposition algorithm, there exists a pair of trees (F, G) with sizes n, m respectively, such that the number of relevant subproblems is $\Omega(m^2n)$.*

PROOF. Let S be the strategy of the decomposition algorithm, and consider the trees F and G depicted in Figure 4. According to Lemma 2.3, every pair (F_v°, G_w°) where $v \in F$ and $w \in G$ is a relevant subproblem for S. Focus on such a subproblem where v and w are internal nodes of F and G. Denote v's right child by v_r and w's left child by w_ℓ. Note that F_v° is a forest whose rightmost root is the node v_r. Similarly, G_w° is a forest whose leftmost root is w_ℓ. Starting from (F_v°, G_w°), consider the computational path $c_{v,w}$ that deletes from F whenever the strategy says left and deletes from G otherwise. In both cases, neither v_r nor w_ℓ is deleted unless one of them is the only node left in the forest. Therefore, the length of this computational path is at least $\min\{|F_v|, |G_w|\} - 1$. Recall that for each subproblem (F', G') along $c_{v,w}$, the rightmost root of F' is v_r and the leftmost root of G' is w_ℓ. It follows that for every two distinct pairs $(v_1, w_1) \neq (v_2, w_2)$ of internal nodes in F and G, the relevant subproblems occurring along the computational paths c_{v_1,w_1} and c_{v_2,w_2} are disjoint. Since there are $\frac{n}{2}$ and $\frac{m}{2}$ internal nodes in F and G respectively, the total number of subproblems along the $c_{v,w}$ computational paths is given by

$$\sum_{(v,w)\ \text{internal nodes}} \min\{|F_v|, |G_w|\} - 1 = \sum_{i=1}^{\frac{n}{2}} \sum_{j=1}^{\frac{m}{2}} \min\{2i, 2j\} = \Omega(m^2n). \quad \square$$

The $\Omega(m^2n)$ lower bound established by Lemma 4.1 is tight if $m = \Theta(n)$, since in this case our algorithm achieves an $O(n^3)$ running time. To establish a tight bound when m is not $\Theta(n)$, we use the following technique for counting relevant subproblems. We associate a subproblem consisting of subforests (F', G') with the unique pair of vertices (v, w) such that F_v, G_w are the smallest trees containing F', G' respectively. For example, for nodes v and w, each with at least two children, the subproblem (F_v°, G_w°) is associated with the pair (v, w). Note that all subproblems encountered in a computational path starting from (F_v°, G_w°) until the point where either forest becomes a tree are also associated with (v, w).

LEMMA 4.2. *For every decomposition algorithm, there exists a pair of trees (F, G) with sizes $n \geq m$ such that the number of relevant subproblems is $\Omega(m^2n \log \frac{n}{m})$.*

PROOF. Consider the trees illustrated in Figure 5. The n-sized tree F is a complete balanced binary tree, and G is a "zigzag" tree of size m. Let w be an internal node of G with a single leaf w_r as its right subtree and w_ℓ as a left child.

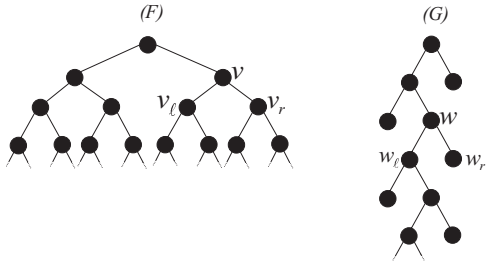

FIG. 5. The two trees used to prove $\Omega(m^2 n \log \frac{n}{m})$ lower bound (Lemma 4.2).

Denote $m' = |G_w|$. Let v be a node in F such that F_v is a tree of size $n' + 1$ where $n' \geq 4m \geq 4m'$. Denote v's left and right children v_ℓ and v_r respectively. Note that $|F_{v_\ell}| = |F_{v_r}| = \frac{n'}{2}$.

Let S be the strategy of the decomposition algorithm. We aim to show that the total number of relevant subproblems associated with (v, w) or with (v, w_ℓ) is at least $\frac{n'}{4}(m' - 2)$. Starting from the subproblem (F_v°, G_w°), which is relevant by Lemma 2.3, let c be the computational path that always deletes from F (no matter whether S says left or right). We consider two complementary cases.

Case 1. *$\frac{n'}{4}$ left deletions occur in the computational path c, and at the time of the $\frac{n'}{4}$th left deletion, there were fewer than $\frac{n'}{4}$ right deletions.*

We define a set of new computational paths $\{c_j\}_{1 \leq j \leq \frac{n'}{4}}$ where c_j deletes from F up through the jth left deletion, and thereafter deletes from F whenever S says right and from G whenever S says left. At the time the jth left deletion occurs, at least $\frac{n'}{4} \geq m' - 2$ nodes remain in F_{v_r} and all $m' - 2$ nodes are present in G_{w_ℓ}. So on the next $m' - 2$ steps along c_j, neither of the subtrees F_{v_r} and G_{w_ℓ} is totally deleted. Thus, we get $m' - 2$ distinct relevant subproblems associated with (v, w). Notice that in each of these subproblems, the subtree F_{v_ℓ} is missing exactly j nodes. So we see that, for different values of $j \in [1, \frac{n'}{4}]$, we get disjoint sets of $m' - 2$ relevant subproblems. Summing over all j, we get $\frac{n'}{4}(m' - 2)$ distinct relevant subproblems associated with (v, w).

Case 2. *$\frac{n'}{4}$ right deletions occur in the computational path c, and at the time of the $\frac{n'}{4}$th right deletion, there were fewer than $\frac{n'}{4}$ left deletions.*

We define a different set of computational paths $\{\gamma_j\}_{1 \leq j \leq \frac{n'}{4}}$ where γ_j deletes from F up through the jth right deletion, and thereafter deletes from F whenever S says left and from G whenever S says right (i.e., γ_j is c_j with the roles of left and right exchanged). Similarly as in case 1, for each $j \in [1, \frac{n'}{4}]$ we get $m' - 2$ distinct relevant subproblems in which F_{v_r} is missing exactly j nodes. All together, this gives $\frac{n'}{4}(m' - 2)$ distinct subproblems. Note that since we never make left deletions from G, the left child of w_ℓ is present in all of these subproblems. Hence, each subproblem is associated with either (v, w) or (v, w_ℓ).

In either case, we get $\frac{n'}{4}(m' - 2)$ distinct relevant subproblems associated with (v, w) or (v, w_ℓ). To get a lower bound on the number of problems we sum over all pairs (v, w) with G_w being a tree whose right subtree is a single node, and

$|F_v| \geq 4m$. There are $\frac{m}{4}$ choices for w corresponding to tree sizes $4j$ for $j \in [1, \frac{m}{4}]$. For v, we consider all nodes of F whose distance from a leaf is at least $\log(4m)$. For each such pair we count the subproblems associated with (v, w) and (v, w_ℓ). So the total number of relevant subproblems counted in this way is

$$\sum_{v,w} \frac{|F_v|}{4}(|G_w| - 2) = \frac{1}{4}\sum_{v} |F_v| \sum_{j=1}^{\frac{m}{4}} (4j - 2)$$

$$= \frac{1}{4}\sum_{i=\log 4m}^{\log n} \frac{n}{2^i} \cdot 2^i \sum_{j=1}^{\frac{m}{4}} (4j - 2) = \Omega\left(m^2 n \log \frac{n}{m}\right). \quad \square$$

THEOREM 4.3. *For every decomposition algorithm and $n \geq m$, there exist trees F and G of sizes $\Theta(n)$ and $\Theta(m)$ such that the number of relevant subproblems is $\Omega(m^2 n(1 + \log \frac{n}{m}))$.*

PROOF. If $m = \Theta(n)$ then this bound is $\Omega(m^2 n)$ as shown in Lemma 4.1. Otherwise, this bound is $\Omega(m^2 n \log \frac{n}{m})$ which was shown in Lemma 4.2. $\square$

5. *Reducing the Space Complexity*

The recursion presented in Section 3 for computing $\delta(F, G)$ translates into an $O(m^2 n(1 + \log \frac{n}{m}))$ time and space algorithm. In this section we reduce the space complexity of this algorithm to $O(mn)$. We achieve this by ordering the relevant subproblems in such a way that we need to record the edit distance of only $O(mn)$ relevant subproblems at any point in time. For simplicity, we assume the input trees F and G are binary. At the end of this section, we show how to remove this assumption.

The algorithm TED fills a global n by m table Δ with values $\Delta_{vw} = \delta(F_v^\circ, G_w^\circ)$ for all $v \in F$ and $w \in G$.

TED(F, G)

1: If $|F| < |G|$ do TED(G, F).
2: For every $v \in$ TopLight(F) do TED(F_v, G).
3: Fill Δ_{vw} for all $v \in$ HeavyPath(F) and $w \in G$.

Step 3 runs in $O(|F||G|^2)$ time and assumes Δ_{vw} has already been computed in step 2 for all $v \in F -$ HeavyPath(F) and $w \in G$ (see Section 3). In the remainder of this section we prove that it can be done in $O(|F||G|)$ space.

In step 3 we go through the nodes $v_1, \ldots, v_t$ on the heavy path of F starting with the leaf v_1 and ending with the root v_t where $t = |\text{HeavyPath}(F)|$. Throughout the computation we maintain a table T of size $|G|^2$. When we start handling v_p $(1 \leq p \leq t)$, the table T holds the edit distance between $F_{v_{p-1}}$ and all possible subforests of G. We use these values to calculate the edit distance between F_{v_p} and all possible subforests of G and store the newly computed values back into T. We refer to the process of updating the entire T table (for a specific v_p) as a period. Before the first period, in which F_{v_1} is a leaf, we set T to hold the edit distance between $\emptyset$ and G' for all subforests G' of G (this is just the cost of deleting G').

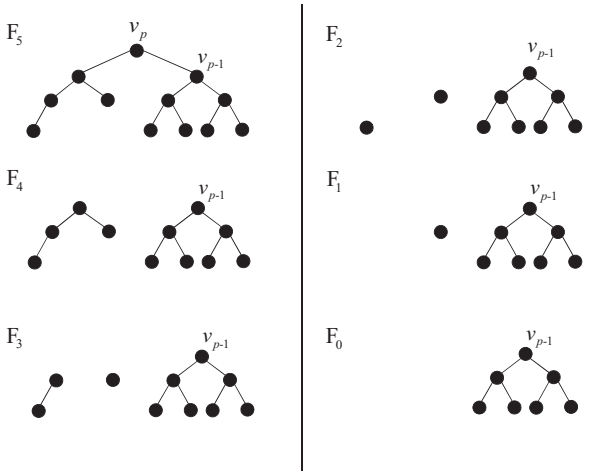

FIG. 6. The *intermediate left subforest enumeration* with respect to $F_{v_{p-1}}$ and F_{v_p} is the sequence of forests $F_{v_{p-1}} = F_0, F_1, \ldots, F_5 = F_{v_p}$.

Note that since we assume F is binary, during each period the direction of our strategy does not change. Let left(v) and right(v) denote the left and right children of a node v. If $v_{p-1} = \text{right}(v_p)$, then our strategy is left throughout the period of v_p. Otherwise it is right. We now explain what goes into computing a period. This process, which we refer to as COMPUTEPERIOD(v_p), both uses and updates tables T and Δ. At the heart of this procedure is a dynamic program. Throughout this description we assume that our strategy is left. The right analog is obvious. We now describe two simple subroutines that are called by COMPUTEPERIOD (v_p).

If $F_{v_{p-1}}$ can be obtained from F_{v_p} by a series of left deletions, the *intermediate left subforest enumeration* with respect to $F_{v_{p-1}}$ and F_{v_p} is the sequence $F_{v_{p-1}} = F_0, F_1 \ldots, F_k = F_{v_p}$ such that $F_{k'-1} = F_{k'} - \ell_{F_{k'}}$ for all $1 \le k' \le k = |F_{v_p}| - |F_{v_{p-1}}|$. This concept is illustrated in Figure 6. The subroutine INTERMEDIATELEFTSUBFORESTENUM($F_{v_{p-1}}, F_{v_p}$) associates every $F_{k'}$ with $\ell_{F_{k'}}$ and lists them in the order of the intermediate left subforest enumerations with respect to $F_{v_{p-1}}$ and F_{v_p}. This is the order in which we access the nodes and subforests during the execution of COMPUTEPERIOD(v_p), so each access will be done in constant time. The intermediate left and right subforest enumerations required for all periods (i.e., for all of the v_ps along the heavy path) can be prepared once in $O(|F|)$ time and space by performing $|F|$ deletions on F according to our strategy and listing the deleted vertices in reverse order.

Let $w_0, w_1, \ldots, w_{|G|-1}$ be the right-to-left preorder traversal of a tree G. We define $G_{i,0}$ as the forest obtained from G by making i right deletions. Notice that the rightmost tree in $G_{i,0}$ is G_{w_i} (the subtree of G rooted at w_i). We further define $G_{i,j}$ as the forest obtained from G by first making i right deletions and then making j left deletions. Let $j(i)$ be the number of left deletions required to turn $G_{i,0}$ into the tree G_{w_i}. We can easily compute $j(0), \ldots, j(|G|-1)$ in $O(|G|)$ time and space by noticing that $j(i) = |G| - i - |G_{w_i}|$. Note that distinct nonempty subforests of G are represented by distinct $G_{i,j}$s for $0 \le i \le |G| - 1$ and $0 \le j \le j(i)$. For convenience, we sometimes refer to $G^\circ_{w_i}$ as $G_{i,j(i)+1}$ and sometimes as the

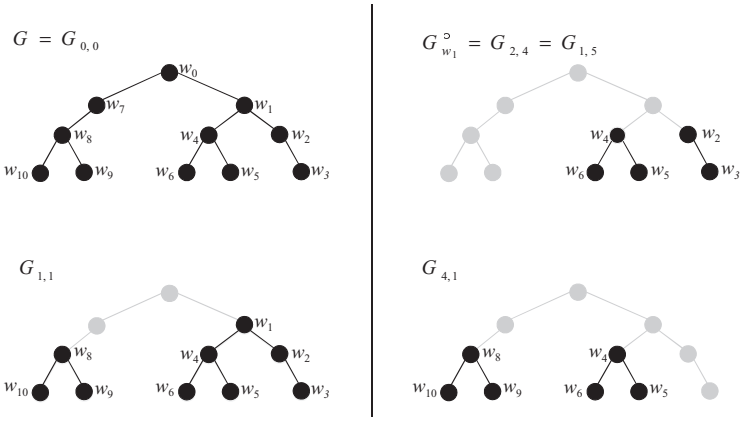

FIG. 7. The indexing of various subforests (shown in solid black) of G (shown in gray). The right-to-left preorder traversal of G is $w_0, w_1, \ldots, w_{|G|-1}$. The subforest $G_{i,j}$ is the forest obtained from G by first making i right deletions and then making j left deletions. All nonempty subforests of G are captured by all $0 \le i \le |G| - 1$ and $0 \le j \le j(i) = |G| - i - |G_{w_i}|$. The index of G itself is $G_{0,0}$. In the special case of $G^{\circ}_{w_1} = G_{2,4}$ we sometimes use the equivalent index $G_{1,5}$.

equivalent $G_{i+1,j(i)}$. The two subforests, are the same since the forest $G_{i,j(i)}$ is the tree G_{w_i}, so making another left deletion, namely $G_{i,j(i)+1}$ is the same as first making an extra right deletion, namely $G_{i+1,j(i)}$. The *left subforest enumeration* of all nonempty subforests of G is defined as

$$G_{|G|-1,j(|G|-1)}, \ldots, G_{|G|-1,0}\ ,\ \ldots\ ,\ G_{2,j(2)}, \ldots, G_{2,0}\ ,\ G_{1,j(1)}, \ldots, G_{1,0}\ ,\ G_{0,0}.$$

The subroutine LEFTSUBFORESTENUM(G) associates every $G_{i,j}$ with the left deleted vertex $\ell_{G_{i,j}}$ and lists them in the order of the left subforest enumeration with respect to G, so that we will be able to access $\ell_{G_{i,j}}$ in this order in constant time per access. This procedure takes $O(|G|)$ time and space for each i by performing first i right deletions and then j left deletions, and listing the left deleted vertices in reverse order. The entire subroutine therefore requires $O(|G|^2)$ time and space. The previous definitions are illustrated in Figure 7. There are obvious "right" analog of everything we have just defined.

The pseudocode for COMPUTEPERIOD(v_p) is given shortly. As we already mentioned, at the beginning of the period for v_p, the table T stores the distance between $F_{v_{p-1}}$ and all subforests of G and our goal is to update T with the distance between F_{v_p} and any subforest of G. For each value of i in decreasing order (the loop in line 3), we compute a temporary table S of the distances between the forests $F_{k'}$ in the intermediate left subforest enumeration with respect to $F_{v_{p-1}}$ and F_{v_p} and the subforest $G_{i,j}$ for $0 \le j \le j(i)$ in the left subforest enumeration of G. Clearly, there are $O(|F||G|)$ such subproblems. The computation is done for increasing values of k' and decreasing values of j according to the basic relation in line 4. Once the entire table S is computed, we update T, in line 5, with the distances between $F_k = F_{v_p}$ and $G_{i,j}$ for all $0 \le j \le j(i)$. Note that along this computation we encounter the subproblem which consists of the root-deleted-trees $F^{\circ}_{v_p} = F_{k-1}$ and $G^{\circ}_{w_{i-1}} = G_{i,j(i-1)}$. In line 7, we store the value for this subproblem in $\Delta_{v_p,w_{i-1}}$. Thus, going over all possible values for i, the procedures updates the entire table T and all the appropriate entries in Δ, and completes a single period.

COMPUTEPERIOD(v_p)

Overwrites T with values $\delta(F_{v_p}, G')$ for all subforests G' of G, and fills in Δ with values $\delta(F^\circ_{v_p}, G^\circ_w)$ for every $w \in G$.

Assumes T stores $\delta(F_{v_{p-1}}, G')$ for all subforests G' of G, and $v_{p-1} = \text{right}(v_p)$ (if $v_{p-1} = \text{left}(v_p)$ then reverse roles of "left" and "right" below).

1: $F_0, \ldots, F_k \leftarrow \text{IntermediateLeftSubforestEnum}(F_{v_{p-1}}, F_{v_p})$

2: $G_{|G|-1,j(|G|-1)}, \ldots, G_{0,0} \leftarrow \text{LeftSubforestEnum}(G)$

3: **for** $i = |G| - 1, \ldots, 0$ **do**

4: compute table $S \leftarrow (\delta(F_{k'}, G_{i,j}))_{\substack{k'=1,\ldots k \\ j=j(i),\ldots,0}}$ via the dynamic program:

$$\delta(F_{k'}, G_{i,j}) = \min \begin{cases} c_{\text{del}}(\ell_{F_{k'}}) + \delta(F_{k'-1}, G_{i,j}), \\ c_{\text{del}}(\ell_{G_{i,j}}) + \delta(F_{k'}, G_{i,j+1}), \\ c_{\text{match}}(\ell_{F_{k'}}, \ell_{G_{i,j}}) + \delta(L^\circ_{F_{k'}}, L^\circ_{G_{i,j}}) \\ \quad +\delta(F_{k'} - L_{F_{k'}}, G_{i,j} - L_{G_{i,j}}) \end{cases}$$

5: $T \leftarrow \delta(F_{v_p}, G_{i,j})$ for all $0 \le j \le j(i)$, via S

6: $Q \leftarrow \delta(F_{k'}, G_{i,j(i-1)})$ for all $1 \le k' \le k$, via S

7: $\Delta \leftarrow \delta(F^\circ_{v_p}, G_{i,j(i-1)})$ via S

8: **end do**

The correctness of COMPUTEPERIOD(v_p) follows from Lemma 2.1. However, we still need to show that all the required values are available when needed in the execution of line 4. Let us go over the different subproblems encountered during this computation and show that each of them is available when required along the computation.

$\delta(F_{k'-1}, G_{i,j})$:

—when $k' = 1$, F_0 is $F_{v_{p-1}}$, so it is already stored in T from the previous period.

—for $k' > 1$, $\delta(F_{k'-1}, G_{i,j})$ was already computed and stored in S, since we go over values of k' in increasing order.

$\delta(F_{k'}, G_{i,j+1})$:

—when $j = j(i)$ and $i + j(i) = |G| - 1$, then $G_{i,j(i)+1} = \emptyset$ so $\delta(F_{k'}, \emptyset)$ is the cost of deleting $F_{k'}$, which may be computed in advance for all subforests within the same time and space bounds.

—when $j = j(i)$ and $i + j(i) < |G| - 1$, recall that $\delta(F_{k'}, G_{i,j(i)+1})$ is equivalent to $\delta(F_{k'}, G_{i+1,j(i)})$ so this problem was already computed, since we loop over the values of i in decreasing order. Furthermore, this problem was stored in the the array Q when line 6 was executed for the previous value of i.

—when $j < j(i)$, $\delta(F_{k'}, G_{i,j+1})$ was already computed and stored in S, since we go over values of j in decreasing order.

$\delta(L^\circ_{F_{k'}}, L^\circ_{G_{i,j}})$:

—this value was computed previously (in step 2 of TED) as Δ_{vw} for some $v \in F - \text{HeavyPath}(F)$ and $w \in G$.

$\delta(F_{k'} - L_{F_{k'}}, G_{i,j} - L_{G_{i,j}})$:

—if $j \neq j(i)$ then $F_{k'} - L_{F_{k'}} = F_{k''}$ where $k'' = k' - |L_{F_{k'}}|$ and $G_{i,j} - L_{G_{i,j}} = G_{i,j'}$ where $j' = j + |L_{G_{i,j}}|$, so $\delta(F_{k''}, G_{i,j'})$ was already computed and stored in S earlier in the loop.

—if $j = j(i)$, then $G_{i,j}$ is a tree, so $G_{i,j} = L_{G_{i,j}}$. Hence, $\delta(F_{k'} - L_{F_{k'}}, G_{i,j} - L_{G_{i,j}})$ is simply the cost of deleting $F_{k''}$.

The space required by this algorithm is evidently $O(|F||G|)$ since the size of S is at most $|F||G|$, the size of T is at most $|G|^2$, the size of Q is at most $|F|$, and the size of Δ is $|F||G|$. The time complexity does not change, since we still handle each relevant subproblem exactly once, in constant time per relevant subproblem.

Note that in the last time COMPUTEPERIOD() is called, the table T stores (among other things) the edit distance between the two input trees. In fact, our algorithm computes the edit distance between any subtree of F and any subtree of G. We could store these values without changing the space complexity.

This concludes the description of our $O(mn)$ space algorithm. All that remains to show is why we may assume the input trees are binary. If they are not binary, we construct in $O(m + n)$ time binary trees F' and G' where $|F'| \leq 2n$, $|G'| \leq 2m$, and $\delta(F, G) = \delta(F', G')$ using the following procedure: Pick a node $v \in F$ with $k > 2$ children which are, in left to right order, $\text{left}(v) = v_1, v_2, \ldots, v_k = \text{right}(v)$. We leave $\text{left}(v)$ as it is, and set $\text{right}(v)$ to be a new node with a special label ε whose children are $v_2, v_3, \ldots, v_k$. To ensure this does not change the edit distance, we set the cost of deleting ε to zero, and the cost of relabeling ε to ∞. The same procedure is applied to G as well. We note that another way to remove the binary trees assumption is to modify COMPUTEPERIOD() to work directly with nonbinary trees at the cost of slightly complicating it. This can be done by splitting it into two parts, where one handles left deletions and the other right deletions.

6. *Conclusions*

We presented a new $O(n^3)$-time and $O(n^2)$-space algorithm for computing the tree edit distance between two rooted ordered trees. Our algorithm is both symmetric in its two inputs as well as adaptively dependent on them. These features make it faster than all previous algorithms in the worst case. Furthermore, we proved that our algorithm is optimal within the broad class of decomposition strategy algorithms, by improving the previous lower bound for this class. As a consequence, any future improvements in terms of worst-case time complexity would have to find an entirely new approach. Our algorithm is simple to describe and implement; our implementation in Python spans just a few dozen lines of code.

ACKNOWLEDGMENTS. We thank the anonymous referees for many helpful comments.

REFERENCES

APOSTOLICO, A., AND GALIL, Z., Eds. 1997. *Pattern Matching Algorithms*. Oxford University Press, Oxford, UK.

BILLE, P. 2005. A survey on tree edit distance and related problems. *Theor. Comput. Sci. 337*, 217–239.

CHAWATHE, S. S. 1999. Comparing hierarchical data in external memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*. 90–101.

CHEN, W. 2001. New algorithm for ordered tree-to-tree correction problem. *J. Algor. 40*, 135–158.

DEMAINE, E. D., MOZES, S., ROSSMAN, B., AND WEIMANN, O. 2007. An optimal decomposition algorithm for tree edit distance. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*. 146–157.

DULUCQ, S., AND TOUZET, H. 2003. Analysis of tree edit distance algorithms. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 83–95.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

HAREL, D., AND TARJAN, R. E. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput. 13*, 2, 338–355.

KLEIN, P. N. 1998. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA)*. 91–102.

KLEIN, P. N., TIRTHAPURA, S., SHARVIT, D., AND KIMIA, B. B. 2000. A tree-edit-distance algorithm for comparing simple, closed shapes. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 696–704.

MOORE, P. 1999. Structural motifs in RNA. *Ann. Rev. Biochem. 68*, 287–300.

SELKOW, S. 1977. The tree-to-tree editing problem. *Inf. Process. Lett. 6*, 6, 184–186.

SHASHA, D., AND ZHANG, K. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput. 18*, 6, 1245–1262.

SHASHA, D., AND ZHANG, K. 1990. Fast algorithms for the unit cost editing distance between trees. *J. Algor. 11*, 4, 581–621.

SLEATOR, D. D., AND TARJAN, R. E. 1983. A data structure for dynamic trees. *J. Comput. Syst. Sci. 26*, 362–391.

TAI, K. 1979. The tree-to-tree correction problem. *J. Assoc. Comput. Mach. 26*, 3, 422–433.

VALIENTE, G. 2002. *Algorithms on Trees and Graphs*. Springer-Verlag.

WAGNER, R. A., AND FISCHER, M. J. 1974. The string-to-string correction problem. *J. ACM 21*, 1, 168–173.

WATERMAN, M. 1995. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapters 13,14. Chapman and Hall.

ZHANG, K. 1995. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recogn. 28*, 3, 463–474.

RECEIVED MAY 2007; REVISED NOVEMBER 2007; ACCEPTED FEBRUARY 2008