An Optimal Decomposition Algorithm for Tree Edit Distance

Erik D. Demaine, Shay Mozes[⋆], Benjamin Rossman, and Oren Weimann

MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar Street, Cambridge, MA 02139, USA.
edemaine@mit.edu,shaymozes@gmail.com,brossman@mit.edu,oweimann@mit.edu

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 paper, we present a worst-case $O(n^3)$-time algorithm for this problem, improving the previous best $O(n^3 \log n)$-time algorithm [7]. 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 $\Omega(n^2 \log^2 n)$ [4] to $\Omega(n^3)$, matching our algorithm's running time. Furthermore, we obtain matching upper and lower bounds of $\Theta(nm^2(1+\log \frac{n}{m}))$ when the two trees have sizes m and n where $m < n$.

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 [2, 3, 8, 11, 13]. One major application is the analysis of RNA molecules in computational biology. The secondary structure of RNA, which plays a significant role in its biological function [9], is naturally represented as an ordered rooted tree [5, 16]. Computing the similarity between the secondary structure of two RNA molecules therefore helps 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 [13] as a generalization of the well-known string edit distance problem [15]. 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

[⋆] Work conducted while visiting MIT

illustrated in Fig. 1. The cost of 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 relabels in both trees that transform F and G into isomorphic trees.

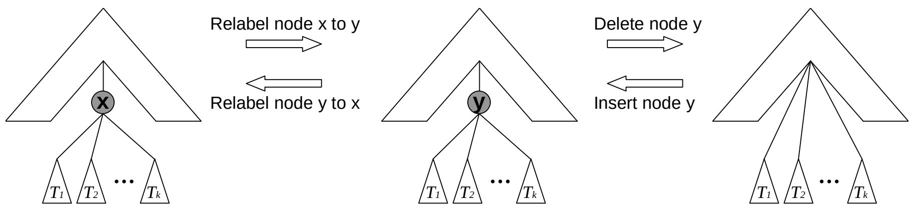

Fig. 1. The three editing operations on a tree with vertex labels.

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 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 [13] presented the first algorithm for computing tree edit distance, which requires $O(n_{\text{leaves}}^2 m_{\text{leaves}}^2 nm)$ time and space, and thus has a worst-case running time of $O(n^3m^3) = O(n^6)$. Shasha and Zhang [11] 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 [7] improved this result to a worst-case $O(m^2 n \log n) = O(n^3 \log n)$ 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.* In [4], Dulucq and Touzet 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 [1, 2, 14] for surveys. 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 [3, 10, 12, 17].

Our results. We present a new algorithm for computing the tree edit distance that falls into the same *decomposition strategy* framework of [4, 7, 11]. 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. 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 [4], and establishes the optimality of our algorithm among all decomposition strategy algorithms. Our algorithm is simple, making it easy to implement, but the analysis of the upper and lower bound proofs is quite complicated.

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. Final conclusions are presented in Section 5. For brevity, some of the proofs and an explicit $O(nm)$ space complexity version of our algorithm will only be presented in the full version of this paper.

2 Background and Framework

Both the existing algorithms and ours compute the edit distance of finite ordered Σ-labeled forests, henceforth *forests*. 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 Σ-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 is denoted F°. The leftmost and rightmost trees of 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 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 v, and we write $c_{\text{match}}(v, w)$ instead of $c_{\text{match}}(\sigma(v), \sigma(w))$ for the cost relabeling v to 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 relabels and deletions of nodes in both forests. In the next two sections, we briefly present the algorithms of Shasha and Zhang, and of Klein. This presentation, inspired by the tree similarity survey of Bille [2], is somewhat different from the original presentations and is essential for understanding our algorithm.

2.1 Shasha and Zhang's Algorithm [11]

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 1 **([11]).** *$\delta(F,G)$ 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 1 yields an $O(m^2n^2)$ dynamic program 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, as we will presently see, 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 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 $\text{cdepth}(v)$ 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 $O(\text{cdepth}(F)\text{cdepth}(G)nm) = O(n_{\text{height}} m_{\text{height}} nm)$ 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 [7]

Klein's algorithm is based on a recursion similar to Lemma 1. Again, we consider forests F and G of sizes $|F| = n \ge |G| = m$. 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* introduced by Harel and Tarjan [6]. Briefly: 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 $\mathrm{ldepth}(v)$ to be the number of light nodes that are ancestors of v in F, and $\mathrm{light}(F)$ as the set of all light nodes in F. By [6], for any forest F and vertex $v \in F$, $\mathrm{ldepth}(v) \leq \log |F| + O(1)$. 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

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

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 and Shasha and Zhang's algorithms are based on Lemma 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 in [4].

Definition 1 (Strategy, Decomposition Algorithm). *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 1 with the directions chosen according to a specific strategy.*

Each strategy is associated with a specific set of recursive calls (or a dynamic program 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 proved a lower bound of $\Omega(mn \log m \log n)$ time for any decomposition strategy algorithm.

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 $O(nm^2(1 + \log \frac{n}{m})) =$

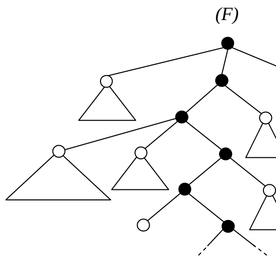

Fig. 2. A tree F with n nodes. Black nodes belong to the heavy path. White nodes are in TopLight(F). The size of each subtree rooted at a white node is at most $\frac{n}{2}$.

$O(n^3)$ running time in the worst case. For clarity we describe the algorithm recursively and analyze its time complexity. In the full version of this paper we prove that the space complexity can be made $O(mn) = O(n^2)$.

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.

For a tree F of size n, define the set TopLight(F) to be the set of roots of the forest obtained by removing the heavy path of F (i.e., the unique path starting from the root along heavy nodes). 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 Fig. 2. Note that the following two conditions are always satisfied:

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

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

The Algorithm. We compute $\delta(F, G)$ recursively as follows:

(1) *If $|F| < |G|$, compute $\delta(G, F)$ instead. That is, make F the larger forest.*

(2) *Recursively compute $\delta(F_v, G)$ for all $v \in$ TopLight(F). Along the way, $\delta(F^\circ_{v'}, G^\circ_w)$ is computed and stored for all v' not in the heavy path of F and for all $w \in G$.*

(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,*

$S(F', G') = \mathsf{right}$. *However, do not recurse into subproblems that were previously computed in step (2).*

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.

Time Complexity. We show that our algorithm has a worst-case runtime 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. Let $R(F, G)$ denote the number of relevant subproblems encountered by the algorithm in the course of computing $\delta(F, G)$.

In step (2) we compute $\delta(F_v, G)$ for all $v \in \mathrm{TopLight}(F)$. Hence, the number of subproblems encountered in this step is $\sum_{v \in \mathrm{TopLight}(F)} R(F_v, G)$. 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 $v \in \mathrm{TopLight}(F)$, so $\delta(F^\circ_{v'}, G^\circ_w)$ is computed in the course of computing $\delta(F_v, G)$ (we prove this formally in Lemma 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

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

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

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

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

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

Proof. We proceed by induction on $|F| + |G|$. The base of the induction is trivial. For the inductive step there are two symmetric cases. If $|F| \geq |G|$ then $R(F, G) \leq$

$|G|^2|F| + \sum_{v\in\mathrm{TopLight}(F)} R(F_v, G)$. Hence, by the inductive assumption,

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

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

This crude estimate gives a worst-case runtime 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:

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

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 $v \in \mathrm{TopLight}(F)$. For each $v \in \mathrm{TopLight}(F)$, notice that v is either in A or in B; it is in A if $|F_v| \ge m$, 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 $u \in \mathrm{TopLight}(F_v)$).

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 $\mathrm{depth}_A(v)$ to be the number of ancestors of v that lie in the set A. We claim that $\mathrm{depth}_A(v) \leq 1 + \log \frac{n}{m}$ 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 $k \leq \log \frac{n}{m}$, i.e., A contains no sequence of descendants of length $> 1 + \log \frac{n}{m}$. So clearly every $v \in F$ has $\mathrm{depth}_A(v) \leq 1 + \log \frac{n}{m}$.

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

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

The relevant subproblems of type (ii) are counted by $\sum_{b\in B} R(F_b, G)$. Using Lemma 2, 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 $m^2 n(1+\log \frac{n}{m})+4m^2 n = O(m^2 n(1+\log \frac{n}{m}))$. This implies:

Theorem 1. *The runtime 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 runtime 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 [4].

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”.

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

Lemma 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.*

The proofs of Lemmas 3 and 4 are given in the full version of this paper. Lemma 4 establishes an $\Omega(m^2 n)$ lower bound on the number of relevant subproblems for any strategy.

Lemma 4. *For any decomposition algorithm, there exists a pair of trees (F, G) with sizes n, m respectively, s.t. the number of relevant subproblems is $\Omega(m^2 n)$.*

This lower bound is tight when $m = \Theta(n)$, since in this case our algorithm achieves an $O(n^3)$ runtime. 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 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 5. *For every decomposition algorithm, there exists a pair of trees (F, G) with sizes $n \geq m$ s.t. the number of relevant subproblems is $\Omega(m^2 n \log \frac{n}{m})$.*

Proof. Consider the trees illustrated in Fig. 3. 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 node w_r as its right subtree and w_ℓ as a left child. 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 $\frac{n'}{4}(m' - 2)$. Let c be the computational path that always deletes from F (no matter whether S says left or right). We consider two complementary cases.

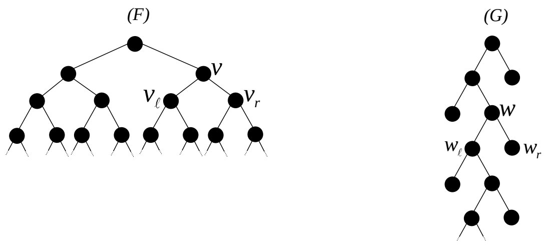

Fig. 3. The two trees used to prove $\Omega(m^2 n \log \frac{n}{m})$ lower bound.

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 $\{c_j\}_{1 \le j \le \frac{n'}{4}}$ 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 $\frac{n'}{4} \ge m' - 2$ 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 $\{\gamma_j\}_{1 \le j \le \frac{n'}{4}}$ 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 $|F_v| \ge 4m$. 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

$$\sum_{v,w} \frac{|F_v|}{4}(|G_w| - 2) = \frac{1}{4}\sum_{v} |F_v| \sum_{j=1}^{\frac{m}{4}} (4j - 2) = \frac{1}{4}\sum_{i=\log 4m}^{\log n} \frac{n}{2^i} \cdot 2^i \sum_{j=1}^{\frac{m}{4}} (4j - 2)$$
$$= \Omega(m^2 n \log \frac{n}{m}) \qquad \square$$

Theorem 2. *For every decomposition algorithm and $n \ge m$, there exist trees F and G of sizes $\Theta(n)$ and $\Theta(m)$ s.t. 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. Otherwise, this bound is $\Omega(m^2 n \log \frac{n}{m})$ which was shown in Lemma 5. $\square$

5 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. Our algorithm is simple to describe and implement; our implementation in Python spans just a few dozen lines of code.

References

1. A. Apostolico and Z. Galil, editors. *Pattern matching algorithms.* Oxford University Press, Oxford, UK, 1997.
2. P. Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337:217–239, 2005.
3. S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 90–101, Edinburgh, Scotland, U.K., 1999.
4. S. Dulucq and H. Touzet. Analysis of tree edit distance algorithms. In *Proceedings of the 14th annual symposium on Combinatorial Pattern Matching (CPM)*, pages 83–95, 2003.
5. D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 1997.
6. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.
7. P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th annual European Symposium on Algorithms (ESA)*, pages 91–102, 1998.
8. P. N. Klein, S. Tirthapura, D. Sharvit, and B. B. Kimia. A tree-edit-distance algorithm for comparing simple, closed shapes. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 696–704, 2000.
9. P.B Moore. Structural motifs in RNA. *Annual review of biochemistry*, 68:287–300, 1999.
10. S.M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, 1977.
11. D. Shasha and K. Zhang. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.
12. D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11(4):581–621, 1990.
13. K. Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery (JACM)*, 26(3):422–433, 1979.
14. G. Valiente. *Algorithms on Trees and Graphs.* Springer-Verlag, 2002.
15. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.
16. M.S. Waterman. *Introduction to computational biology: maps, sequences and genomes,* chapters 13,14. Chapman and Hall, 1995.
17. K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 28(3):463–474, 1995.