Faster Algorithms for Bounded Tree Edit Distance

Shyan Akmal
MIT, EECS and CSAIL, Cambridge, MA, USA

Ce Jin
MIT, EECS and CSAIL, Cambridge, MA, USA

Abstract

Tree edit distance is a well-studied measure of dissimilarity between rooted trees with node labels. It can be computed in $O(n^3)$ time [Demaine, Mozes, Rossman, and Weimann, ICALP 2007], and fine-grained hardness results suggest that the weighted version of this problem cannot be solved in truly subcubic time unless the APSP conjecture is false [Bringmann, Gawrychowski, Mozes, and Weimann, SODA 2018].

We consider the *unweighted* version of tree edit distance, where every insertion, deletion, or relabeling operation has unit cost. Given a parameter k as an upper bound on the distance, the previous fastest algorithm for this problem runs in $O(nk^3)$ time [Touzet, CPM 2005], which improves upon the cubic-time algorithm for $k \ll n^{2/3}$. In this paper, we give a faster algorithm taking $O(nk^2 \log n)$ time, improving both of the previous results for almost the full range of $\log n \ll k \ll n/\sqrt{\log n}$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases tree edit distance, edit distance, dynamic programming

Digital Object Identifier 10.4230/LIPIcs.ICALP.2021.12

Category Track A: Algorithms, Complexity and Games

Funding *Shyan Akmal*: Supported by NSF Grant CCF-1909429.
Ce Jin: Supported by an MIT Akamai Presidential Fellowship.

Acknowledgements We thank Virginia Vassilevska Williams for several helpful discussions.

1 Introduction

Many tasks involve measuring the similarity between two sets of data. When the data is naturally represented as a string of characters, one of the most popular and well-studied ways of measuring similarity is via the (string) edit distance, defined to be the minimum number of characters that must be deleted, inserted, and substituted to turn one string into the other. Although edit distance is a fundamental problem in computer science and has been employed to great effect in many other areas, it can be less useful for applications where we are interested in comparing data that is not just linearly ordered, but has some hierarchical organization. When the data admits a tree structure, a natural measure of similarity is the *tree edit distance*, first introduced by Tai [34] as a generalization of the string edit distance problem [38]. Computing this metric has a wide variety of applications in a diverse array of fields including computational biology [22, 32, 23, 39], structured data analysis [14, 16, 21], and image processing [7, 26, 25, 31].

Given two *rooted ordered* trees with node labels, the tree edit distance is the minimum number of node deletions, insertions, and relabelings needed to turn one tree into the other. When we delete a node, its children become children of the parent of the deleted node. Beyond this widely studied definition, there are many other variants of the tree edit distance problem, including those defined for unrooted trees or unordered trees, or parameterized by the depth or the number of leaves, which we do not consider in this paper. We refer interested readers to the survey by Bille [8] for a comprehensive review.

We now recount the development of exact algorithms for tree edit distance. In 1979, Tai [34] gave the first algorithm that computes the tree edit distance between two node-labeled rooted trees on n nodes in $O(n^6)$ time. The time complexity was improved to $O(n^4)$ by Zhang and Shasha [40] using a dynamic programming approach. Later, Klein [24] applied the heavy-light decomposition technique to obtain an $O(n^3 \log n)$ time algorithm. Finally, Demaine, Mozes, Rossman, and Weimann [17] improved the running time by a log-factor to $O(n^3)$, and further showed that this running time is optimal among a certain class of dynamic programming algorithms termed *decomposition strategy algorithms* by Dulucq and Touzet [19, 20]. When the two input trees have different sizes $m \le n$, their algorithm runs in $O\left(nm^2(1 + \log \frac{n}{m})\right)$ time.

All algorithms mentioned above actually compute tree edit distance in the general *weighted* setting where the cost of deleting, inserting, or relabeling is a function of the labels (so that deleting nodes with certain labels might be cheaper than deleting other nodes with different labels). In this setting, Bringmann, Gawrychowski, Mozes, and Weimann [13] showed conditional hardness results for the tree edit distance problem: a truly subcubic time algorithm for this problem would imply a truly subcubic time algorithm for the All-Pairs Shortest Paths (APSP) problem (assuming alphabet of size $\Theta(n)$), and an $O(n^{k(1-\varepsilon)})$ time algorithm for the Max-weight k-clique problem (assuming a sufficiently large constant-size alphabet). However, the instances produced by their fine-grained reduction have non-unit edit costs, and it is not clear yet how to prove a conditional hardness result for the *unweighted* tree edit distance problem with unit edit costs. In contrast, the quadratic-time fine-grained lower bound for the *string* edit distance problem (based on the Strong Exponential Time Hypothesis) holds for unit-cost operations [6, 1].

Therefore, it is natural to consider the unweighted unit-cost setting, where every elementary operation has cost 1, independent of the labels. In this case, the distance between two trees of sizes n and m cannot be larger than $n + m$, and is arguably even smaller in practical scenarios. In 2005, Touzet [35, 36] gave an algorithm in this context that computes the unweighted tree edit distance in $O(nk^3)$ time, assuming the distance is at most k. When $k = \Theta(n)$, Touzet's algorithm has the same performance as the $O(n^4)$ time algorithm by Zhang and Shasha [40]. However, the running time significantly improves if the upper bound k is much smaller than n. We remark that similar progress was shown earlier for the string edit distance problem: although the best known running time for the general case is $O(n^2/\log^2 n)$ [29, 9], when the distance is at most k, Ukkonen [37] gave an $O(nk)$ time algorithm, which was later improved to $\widetilde{O}(n + k^2)$ time[1] by Myers [30], Landau and Vishkin [28] using suffix trees.

Although we focus on exact algorithms in this work, approximation algorithms for the tree edit distance problem have also been studied [2, 11]. Boroujeni, Ghodsi, Hajiaghayi, and Seddighin [11] showed an algorithm that computes a $(1 + \varepsilon)$-approximation of the tree edit distance in $\widetilde{O}(\varepsilon^{-3}n^2)$ time. If an upper bound k on the distance is known, the running time can be improved to $\widetilde{O}(\varepsilon^{-3}nk)$. For the easier problem of approximating string edit distance, there is a longer line of research [5, 3, 10, 15, 12, 27] culminating in a near-linear time constant-factor approximation algorithm [4].

[1] In this paper, $\widetilde{O}(f)$ stands for $f \cdot (\log f)^{O(1)}$.

1.1 Our contribution

We present a faster algorithm for exactly computing the unweighted tree edit distance (where every elementary operation has unit cost), with a parameter $k \leq O(n)$ given as an upper bound on the distance.

▶ **Theorem 1.** *Given two node-labeled rooted trees T_1, T_2 each of size at most n, we can compute the unweighted tree edit distance between T_1 and T_2 exactly in $O(nk^2 \log n)$ time, assuming the distance is at most k.*

When the distance parameter k is constant our algorithm runs in quasilinear time, and as k reaches its upper bound $O(n)$ we recover the $O(n^3 \log n)$ time algorithm by Klein [24]. Our algorithm outperforms the $O(n^3)$ time algorithms of Demaine et al. [17] when $k = o(n/\sqrt{\log n})$. As mentioned earlier, the previous best algorithm for bounded tree edit distance by Touzet [36] takes $O(nk^3)$ time. The time complexity of our algorithm improves upon this prior work whenever $k = \omega(\log n)$.

1.2 High-level Overview

Touzet's $O(nk^3)$-time algorithm is based on Zhang and Shasha's $O(n^4)$-time dynamic programming algorithm [40]. The improvement was achieved by pruning unuseful DP states, and only considering $O(nk^3)$ many states instead of $O(n^4)$. This pruning technique was inspired by an idea used in the previous $O(nk)$-time algorithm for string edit distance [37]: for input strings whose edit distance is at most k, when building the dynamic programming table for computing the edit distance, it suffices to only compute entries of the table corresponding to prefixes whose lengths differ by at most k. Touzet's improvement for tree edit distance employs a similar technique and relies on measuring the "distance" between two DP states with respect to the preorder tree traversal, which is compatible with the DP transitions of Zhang and Shasha.

We modify Klein's $O(n^3 \log n)$ time algorithm by further reducing the number of useful states, similar in spirit to the algorithm by Touzet [36]. The main difficulty in adapting this idea is that unlike the algorithm of Zhang and Sasha, Klein's DP algorithm does not follow the same preorder traversal of the nodes. Hence we need completely new arguments to bound the number of useful DP states. Beyond considering the sizes of the subproblems generated, our proofs examine how various subforests are generated by different transition rules and employ some combinatorial arguments about how the subgraphs of deleted nodes can be structured when the edit distance is known to be bounded.

1.3 Paper Organization

In Section 2 we formally define the tree edit distance problem and introduce the notation used throughout the rest of the paper. Next, in Section 3, we review Klein's algorithm [24] which our algorithm builds off of. Then, in Section 4, we present our improved algorithm. Finally, we conclude by mentioning several open questions relevant to our work in Section 5.

2 Preliminaries

In this paper, we consider rooted trees that are *ordered*, meaning that the order between siblings is significant. We also consider *forests* consisting of disjoint rooted trees, where the order between these trees is also significant. It is convenient to treat the tree roots of a forest as the children of a virtual root node. Let $\mathsf{par}(v)$ denote the parent node of v, or the virtual root node if v is a tree root in the forest.

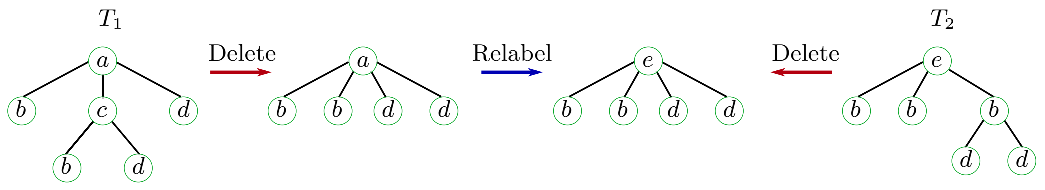

Figure 1 To turn T_1 and T_2 into the same tree with a minimum number of operations, we can delete a node from each and relabel a node in T_1. So in this example $\mathsf{ed}(T_1, T_2) = 3$.

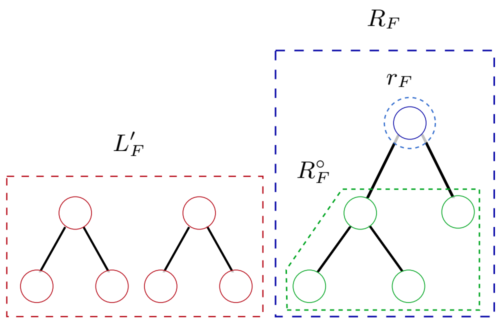

Figure 2 The example forest F above is partitioned into L'_F, r_F, and R°_F.

We define the *node removal operation* in the following natural way: after removing a node v from the forest F, the children of v become children of $\mathsf{par}(v)$, preserving the same relative order. We use $F - v$ to denote the forest obtained by removing v from F.

We now formally define the *tree edit distance* as a metric on ordered rooted trees with node labels.

▶ **Definition 2** ((Unweighted) Tree Edit Distance)**.** *Let T_1 and T_2 be two ordered rooted trees whose nodes are labeled with symbols from some alphabet Σ. There are two types of allowed operations:*

- *Relabeling: change the label of a node from one symbol in Σ to another.*
- *Deletion: remove a node.*

Then the tree edit distance between T_1 and T_2, denoted by $\mathsf{ed}(T_1, T_2)$, is the minimum number of operations that must be performed on T_1 and T_2 to obtain two identical forests.

Figure 1 provides an example of these operations in action.

▶ Remark 3. An alternative definition of tree edit distance is the minimum number of insertions, deletions, and relabeling needed to turn one tree into the other. It is easy to see that these two definitions are equivalent.

Since the operations of relabeling and deletion also apply to labeled forests, the above definition naturally extends to measure the edit distance between two *forests* F_1 and F_2, and for the rest of the paper we write $\mathsf{ed}(F_1, F_2)$ to denote this edit distance as well.

Given a forest F, we write L_F (or R_F) to denote the leftmost (or rightmost) tree in F, and write ℓ_F (or r_F) to denote the root of L_F (or R_F). For convenience, let L'_F denote $F - R_F$, and let R°_F denote $R_F - r_F$ (similarly, $R'_F = F - L_F$ and $L^\circ_F = L_F - \ell_F$). Hence, the nodes of a nonempty forest F can be partitioned into three parts: L'_F, r_F, and R°_F (an example is given in Figure 2). Finally, $\mathsf{size}(F)$ or $|F|$ denote the number of nodes in F (where F can also be any subset of nodes).

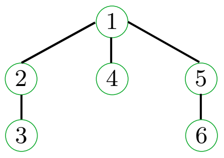

Figure 3 The subforests of this rooted tree are: $\{1,2,3,4,5,6\}$, $\{2,3,4,5,6\}$, $\{3,4,5,6\}$, $\{4,5,6\}$, $\{5,6\}$, $\{6\}$, $\emptyset$, $\{2,3,4,6\}$, $\{3,4,6\}$, $\{4,6\}$, $\{2,3,4\}$, $\{3,4\}$, $\{4\}$, $\{2,3\}$, $\{3\}$. For example, the subforest $\{3,4,6\}$ can be obtained by first removing the leftmost root 1, then removing the rightmost root 5, and finally removing the leftmost root 2.

▶ **Definition 4** (Subforest). *Given a rooted tree T, we say F is a* subforest *of T if we can obtain F from T by repeatedly deleting the leftmost or rightmost root.*

An example illustrating the definition of subforests is given in Figure 3.

▶ **Proposition 5.** *A rooted tree T of n nodes has at most $O(n^2)$ subforests.*

Proof. Although a subforest may result from interleaving operations of removing the leftmost root and removing the rightmost root, it is not hard to see that every such subforest F can also be obtained from T by first removing the leftmost root a times, and then removing the rightmost root b times, for some nonnegative integer a, b with $a + b \le n$. Specifically, let u be the node in F with the smallest index $\mathsf{pre}(u)$ in the *preorder traversal* of T ($1 \le \mathsf{pre}(u) \le n$), and we can set $a = \mathsf{pre}(u) - 1$ and $b = n - a - \mathsf{size}(F)$. The claim then follows from the number of choices of (a, b). ◀

For a subforest F of T, define $\mathsf{LCA}_T(F)$ as the lowest common ancestor in T of all nodes in F. When the identity T is clear from context, we may write $\mathsf{LCA}(F)$ and leave the underlying tree implicit. Observe that $\mathsf{LCA}(F)$ is in F precisely when F is a subtree of T.

Throughout, we use T_1, T_2 to denote the input trees (or T if we do not specify which one of the two) we want to compute the edit distance between.

3 Review of Klein's Algorithm

We briefly review Klein's algorithm [24] in the context of computing the unweighted tree edit distance $\mathsf{ed}(T_1, T_2)$ (see [17, 8, 18] for other overviews of this algorithm).

The algorithm uses dynamic programming (DP) over pairs (F_1, F_2), where F_1, F_2 are subforests of T_1, T_2, respectively. Let the node relabeling cost $\delta(x, y) = 1$ if nodes x, y have different labels, and $\delta(x, y) = 0$ otherwise. Then $\mathsf{ed}(F_1, F_2)$ can be computed recursively as follows [40]:

- The base case is where either of F_1, F_2 is empty (denoted as $\emptyset$), and we have

$$\mathsf{ed}(F_1, \emptyset) = \mathsf{size}(F_1), \mathsf{ed}(\emptyset, F_2) = \mathsf{size}(F_2). \tag{1}$$

- When both F_1, F_2 are nonempty, if $\mathsf{size}(L_{F_1}) > \mathsf{size}(R_{F_1})$, then we recurse with

$$\mathsf{ed}(F_1, F_2) = \min \begin{cases} \mathsf{ed}(F_1 - r_{F_1}, F_2) + 1 \\ \mathsf{ed}(F_1, F_2 - r_{F_2}) + 1 \\ \mathsf{ed}(R^\circ_{F_1}, R^\circ_{F_2}) + \mathsf{ed}(L'_{F_1}, L'_{F_2}) + \delta(r_{F_1}, r_{F_2}). \end{cases} \tag{2}$$

- Otherwise, $\mathsf{size}(L_{F_1}) \leq \mathsf{size}(R_{F_1})$, and we recurse with

$$\mathsf{ed}(F_1, F_2) = \min \begin{cases} \mathsf{ed}(F_1 - \ell_{F_1}, F_2) + 1 \\ \mathsf{ed}(F_1, F_2 - \ell_{F_2}) + 1 \\ \mathsf{ed}(L^\circ_{F_1}, L^\circ_{F_2}) + \mathsf{ed}(R'_{F_1}, R'_{F_2}) + \delta(\ell_{F_1}, \ell_{F_2}). \end{cases} \tag{3}$$

Taking Equation (2) as an example, the recursion considers three options concerning the rightmost roots of F_1, F_2: (1) r_{F_1} is removed. (2) r_{F_2} is removed. (3) The two roots are matched to each other, generating two subproblems of matching their subtrees $R^\circ_{F_1}, R^\circ_{F_2}$, and matching the remaining parts L'_{F_1}, L'_{F_2}. The other recursion rule in Equation (3) is symmetric and considers the leftmost roots.

We can easily verify that, if we compute $\mathsf{ed}(T_1, T_2)$ using this recursion, the DP states visited by the recursion are indeed pairs of subforests of T_1 and T_2. We call a subforest F_1 or F_2 which appears in the above dynamic programming procedure a *relevant* subforest. Klein showed the following bound on the number of relevant subforests F_1 of T_1 generated by the DP procedure.

▶ **Lemma 6** (Lemma 3 of [24]). *If we use top-down dynamic programming to compute* $\mathsf{ed}(T_1, T_2)$ *with respect to the recursion defined in Equations* (1)–(3)*, we only ever need to compute* $\mathsf{ed}(F_1, F_2)$ *for* $O(|T_1| \log |T_1|)$ *distinct subforests* F_1 *of* T_1*.*

The proof of this lemma uses a heavy-light decomposition argument, which crucially relies on choosing the "direction" of recursion (Equations (2) and (3)) based on the sizes of the leftmost and rightmost trees in F_1. This improves upon the previous DP algorithm by Zhang and Shasha [40], which always recurses on the rightmost roots and could only give an $O(|T_1|^2)$ bound instead of $O(|T_1| \log |T_1|)$.

Since there are only $O(|T_2|^2)$ possible subforests F_2 of T_2 (Proposition 5), Lemma 6 shows that we can compute $\mathsf{ed}(T_1, T_2)$ in $O(|T_1||T_2|^2 \log |T_1|)$ time. In the next section, we show how to use the assumption that $\mathsf{ed}(T_1, T_2) \leq k$ to bound the number of relevant F_2 as well, and through this get a faster algorithm.

4 Improved Algorithm

4.1 DP state transition graph

Our algorithm builds on Klein's DP algorithm described in Section 3. For the sake of analysis, it is helpful to consider the DP state transition graph, which is a directed acyclic graph with vertices representing the DP states (F_1, F_2) and edges representing DP transitions. Each edge is associated with a proxy cost that lower bounds the true incurred cost when using this transition in the actual DP. These will be based off the trivial lower bound

$$\mathsf{ed}(F_1, F_2) \geq |\mathsf{size}(F_1) - \mathsf{size}(F_2)|\,, \tag{4}$$

which holds because each operation changes the size of a tree by at most 1, and at the end of applying $\mathsf{ed}(F_1, F_2)$ operations the trees must have the same size.

To define the DP state transition graph, we distinguish three types of DP transition that can occur from following the recursion of Klein's algorithm described in Equations (2) and (3). The first type corresponds to the first two cases of Equations (2) and (3) where we delete the rightmost or leftmost root of the forest. The second and third types of transition capture the two subproblems generated from the third case of Equations (2) and (3) where we match nodes in the trees. Hence, the edges in the DP state transition graph and their proxy costs are defined as follows:

Type 1 (Node Removal) We delete the rightmost (or leftmost) root of F_1 (or F_2).
For example, we can transition $(F_1, F_2) \rightarrow (F_1 - r_{F_1}, F_2)$. This transition has cost 1.

Type 2 (Subtree Removal) We remove the rightmost (or leftmost) subtrees of F_1 and F_2.
For example, we can transition $(F_1, F_2) \rightarrow (L'_{F_1}, L'_{F_2})$. This transition costs at least $|\mathsf{size}(R_{F_1}) - \mathsf{size}(R_{F_2})|$ by Equation (4) and the last case of Equation (2).

Type 3 (Subtree Selection) We focus on the subtrees below the rightmost (or leftmost) roots of F_1 and F_2.
For example, we can transition $(F_1, F_2) \rightarrow (R^\circ_{F_1}, R^\circ_{F_2})$. This transition costs at least $|\mathsf{size}(L'_{F_1}) - \mathsf{size}(L'_{F_2})|$ by Equation (4) and the last case of Equation (2).

4.2 Pruning DP states

Each pair of subforests (F_1, F_2) is a potential state in the DP table. We say a state (F_1, F_2) is "not useful" or *useless* if we do not need to evaluate $\mathsf{ed}(F_1, F_2)$ to compute the overall tree edit distance $\mathsf{ed}(T_1, T_2)$. Having defined the DP state transition graph, we use the following simple observation to label some states as useless.

▶ **Proposition 7** (DP State Pruning Rule 1). *Suppose input trees T_1, T_2 satisfy $\mathsf{ed}(T_1, T_2) \leq k$. Then if a state cannot be reached from (T_1, T_2) by traversing a sequence of edges with total cost at most k in the DP state transition graph, that state is useless.*

We will also make use of the following pruning rule, which is a direct application of Equation (4).

▶ **Proposition 8** (DP State Pruning Rule 2). *Suppose input trees T_1, T_2 satisfy $\mathsf{ed}(T_1, T_2) \leq k$. If $|\mathsf{size}(F_1) - \mathsf{size}(F_2)| > k$, then the DP state (F_1, F_2) is useless.*

The two pruning rules will enable us to prove the following core result, which shows that when the tree edit distance is bounded, each relevant subforest cannot occur in too many useful states.

▶ **Lemma 9** (Number of useful DP states). *Suppose input trees T_1, T_2 satisfy $\mathsf{ed}(T_1, T_2) \leq k$. For each relevant subforest F_1 of T_1, there are at most $O(k^2)$ subforests F_2 of T_2 such that (F_1, F_2) is a useful DP state.*

Lemma 9 together with Lemma 6 immediately shows an $O(nk^2 \log n)$ bound on the number of useful DP states, which will suffice to prove Theorem 1, so in the remainder of this section, we setup the proof of this lemma.

▶ **Definition 10** (Upper parts). *Given a subforest F of T, we partition the nodes of $T \setminus F$ into three disjoint upper parts MU_F, LU_F, and RU_F as follows.*

- *The middle upper part MU_F contains the nodes on the path from the root of T to $\mathsf{LCA}(F)$ (excluding $\mathsf{LCA}(F)$ if $\mathsf{LCA}(F) \in F$).*
- *The left upper part is defined as $LU_F := \{u \in T \setminus MU_F \mid \mathsf{pre}(u) < \mathsf{pre}(v)$ for all $v \in F\}$, where $\mathsf{pre}(u)$ denote the index of u in the preorder traversal of T $(1 \leq \mathsf{pre}(u) \leq |T|)$. The right upper part RU_F is defined symmetrically using the postorder traversal of T. Intuitively, LU_F consists of the nodes to the left of the path MU_F, and RU_F consists of the nodes to the right of this path.*

See Figure 4 for some examples.

If a DP state (G_1, G_2) can be reached from (T_1, T_2) in the DP state transition graph, it means that we obtain G_1 and G_2 by removing some nodes in T_1 and T_2 respectively, following the DP transition rules. We classify the removed nodes in $T_1 \setminus G_1$ according to which of

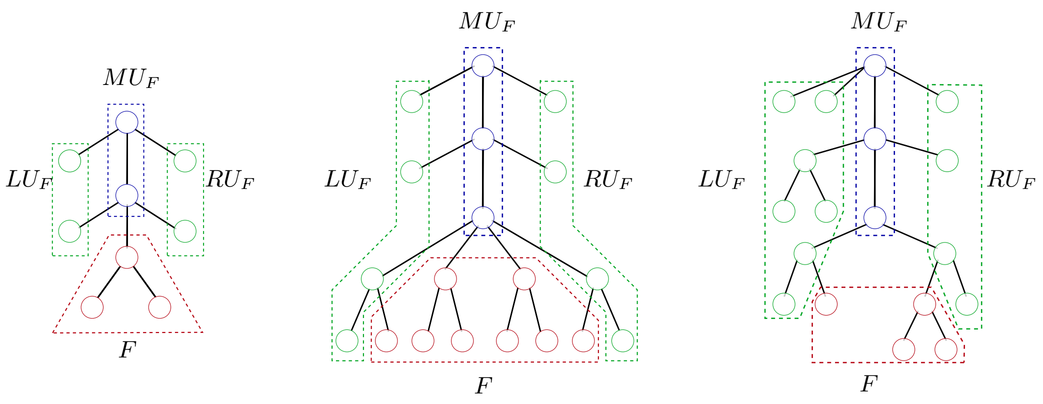

Figure 4 Three examples of subforests F in different underlying trees, with upper parts labeled.

the three upper parts they belong to. For node $v \in T_1 \setminus G_1$, if $v \in LU_{G_1}$ (or $v \in RU_{G_1}$, $v \in MU_{G_1}$), then we say v is *left-removed* (or *right-removed*, *middle-removed*) with respect to subforest G_1. If during a DP transition $(F_1, F_2) \to (G_1, G_2)$, a node $v \in F_1 \setminus G_1$ is left-removed (or right-removed, middle-removed) with respect to not only G_1, but also all subforests $G_1' \subseteq G_1$ (which may be reached by later DP transitions), then we simply say v is left-removed (or right-removed, middle-removed) during this DP transition, without specifying the subforest G_1. The above discussion also similarly applies to the second input tree T_2 and its subforests.

By inspecting the DP transition rules described in Section 4.1, we immediately have the following simple but useful observation.

▶ **Lemma 11.** *Let (F_1, F_2) be a DP state. The following hold:*

- *A type 2 transition from this state either right-removes* $\mathsf{size}(R_{F_1})$ *nodes, or left-removes* $\mathsf{size}(L_{F_1})$ *nodes from F_1, depending on whether the right or left subtree were removed.*
- *A type 3 transition from this state either left-removes* $\mathsf{size}(F_1 - R_{F_1})$ *nodes and middle-removes one node, or right-removes* $\mathsf{size}(F_1 - L_{F_1})$ *nodes and middle-removes one node from F_1, depending on whether the transition zoomed in on the right or left subtree.*

Similar statements hold for removals in F_2.

Note that in the case of type 1 transitions, we cannot tell whether the node being removed was a left, middle, or right-removal. However, we observe that a type 1 transition always has cost 1. Combining this observation with Lemma 11 and the pruning rule in Proposition 7, we obtain the following property of useful DP states (G_1, G_2):

▶ **Lemma 12.** *If DP state (G_1, G_2) survives the pruning rule in Proposition 7, then*

$$|\mathsf{size}(LU_{G_1}) - \mathsf{size}(LU_{G_2})| \le k,$$

and

$$|\mathsf{size}(RU_{G_1}) - \mathsf{size}(RU_{G_2})| \le k.$$

Proof. Consider the sets LU_{G_1} and LU_{G_2} of left-removed nodes in G_1 and G_2. Suppose k_1 nodes of LU_{G_1} and k_2 nodes of LU_{G_2} were removed by type 1 transitions, incurring a total cost of $k_1 + k_2$. The remaining $\mathsf{size}(LU_{G_1}) - k_1$ nodes in LU_{G_1} and $\mathsf{size}(LU_{G_2}) - k_2$ nodes in LU_{G_2} must be the result of type 2 and 3 transitions.

From Lemma 11 and the discussion in Section 4.1, we know that when a type 2 or 3 transition t left-removes $c_1^{(t)}$ nodes from T_1 and $c_2^{(t)}$ nodes from T_2, the incurred cost is at least $|c_1^{(t)} - c_2^{(t)}|$. Then by triangle inequality, the total cost from all type 2 and 3 transitions is at least

$$\sum_t \left|c_1^{(t)} - c_2^{(t)}\right| \geq \left|\sum_t c_1^{(t)} - \sum_t c_2^{(t)}\right| = \left|(\mathsf{size}(LU_{G_1}) - k_1) - (\mathsf{size}(LU_{G_2}) - k_2)\right|,$$

where the sum is over all type 2 and 3 transitions t leading from state (T_1, T_2) to state (G_1, G_2). Then, by applying triangle inequality once more, the total cost from all transitions is at least

$$k_1 + k_2 + \left|(\mathsf{size}(LU_{G_1}) - k_1) - (\mathsf{size}(LU_{G_2}) - k_2)\right| \geq \left|\mathsf{size}(LU_{G_1}) - \mathsf{size}(LU_{G_2})\right|.$$

This proves the first inequality. The second inequality follows from identical reasoning, applied to the right-removed instead of the left-removed nodes of G_1 and G_2. ◀

We have just derived the useful Lemma 12 from the first pruning rule in Proposition 7. To prove Lemma 9, we still need to apply the second pruning rule in Proposition 8 as well. We will use the following lemma.

▶ **Lemma 13.** *Given three integers a, b, c, the number of subforests F of a tree T which simultaneously satisfy $|\mathsf{size}(LU_F) - a| \leq k$, $|\mathsf{size}(RU_F) - b| \leq k$, and $|\mathsf{size}(F) - c| \leq k$ is at most $O(k^2)$.*

Before proving Lemma 13, we show that it implies the desired upper bound on the number of useful DP states that survive both pruning rules in Proposition 7 and Proposition 8.

Proof of Lemma 9 given Lemma 13. We are given a relevant subforest F_1 of T_1, and want to bound the number of subforests F_2 of T_2 such that (F_1, F_2) is a useful state. By Lemma 12, the state (F_1, F_2) is useful only if

$$|\mathsf{size}(LU_{F_2}) - a|, |\mathsf{size}(RU_{F_2}) - b| \leq k$$

for $a = \mathsf{size}(LU_{F_1})$ and $b = \mathsf{size}(RU_{F_1})$. Moreover, by Proposition 8 if the state is useful then

$$|\mathsf{size}(F_2) - c| \leq k$$

for $c = \mathsf{size}(F_1)$. Hence, applying Lemma 13 with $T = T_2$ immediately implies that there are $O(k^2)$ possibilities for F_2, which proves the desired result. ◀

4.3 Proof of Lemma 13

Suppose $\mathsf{size}(LU_F) = \ell$ and $\mathsf{size}(RU_F) = r$ for some integers ℓ and r within k of a and b respectively. Then we claim the following algorithm outputs all possible subforests F satisfying the hypotheses of the lemma:

```
1. Initialize $F = T$ as the given tree.
2. While $\ell \neq 0$ or $r \neq 0$:
   a. If $F$ has only root remaining, delete this root (middle-removal) from $F$
   b. Otherwise, $F$ has more than one root remaining:
      i. If $\mathsf{size}(L_F) \leq \ell$: remove the leftmost tree and update $\ell \leftarrow \ell - \mathsf{size}(L_F)$
      ii. Else if $\mathsf{size}(R_F) \leq r$: remove the rightmost tree and update $r \leftarrow r - \mathsf{size}(R_F)$
      iii. Otherwise remove the leftmost root $\ell$ times, remove the rightmost root $r$ times, and
           return $F$ (unique solution case)
```

3. If F has one root remaining: repeatedly remove the only root (middle-removal) until we no longer have a single root. Return all the forests encountered during this procedure as possible solutions of F *(multiple solutions case)*
4. Otherwise, F has more than one root remaining: return F *(unique solution case)*

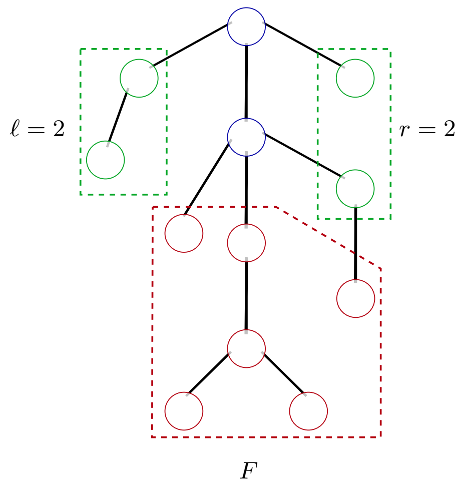

Figure 5 An example of the unique solution case, where $\ell = 2$ and $r = 2$.

At each step of the algorithm, we are either at a state with multiple roots or at a state with one root. In the former case, we have to left-remove or right-remove which we do (unless we have already left-removed ℓ times and right-removed r times, in which case we halt). In the latter case, if we still have not left-removed or right-removed the full number of times, we must keep middle-removing until we can make left or right removals. Terminating in one of these states corresponds to the *unique solution* cases of the algorithm (at step 2(b)iii or step 4). An example is given in Figure 5.

In these situations, the algorithm halts on the unique subforest F of T with $\mathsf{size}(LU_F) = \ell$ and $\mathsf{size}(RU_F) = r$. Since there are $O(k)$ possible values for ℓ and r individually, we get that there are at most $O(k^2)$ distinct subforests F which can be outputted as a "unique solution" in the above procedure.

The only other possibility is that we find ourselves in step 3 of the algorithm at a point where we have already left-removed ℓ times and right-removed r times, and there is only one root u remaining. In this case F might not be uniquely determined: we can continue to middle-remove the remaining root for some number of times and then return a possible solution of F. Formally, let w be the deepest descendant of the remaining root u, such that for every node v on the path from w to u, v has no siblings. Then, for every such node v, the subtree rooted at v (denoted T_v) and $T_v - v$ can be a valid solution for F. This describes the *multiple solutions case* annotated in step 3 of the above procedure. An example of the multiple solutions case is given in Figure 6.

By the above discussion, a subforest F from the multiple solutions case can be determined uniquely by the identity of the lowest common ancestor $v = \mathsf{LCA}(F)$, and the choice of whether v is in F or not. We now prove that, over all choices of valid ℓ and r, there are only $O(k)$ many possibilities for the node v. Combined with the unique solution case, this will immediately finish the proof of the lemma.

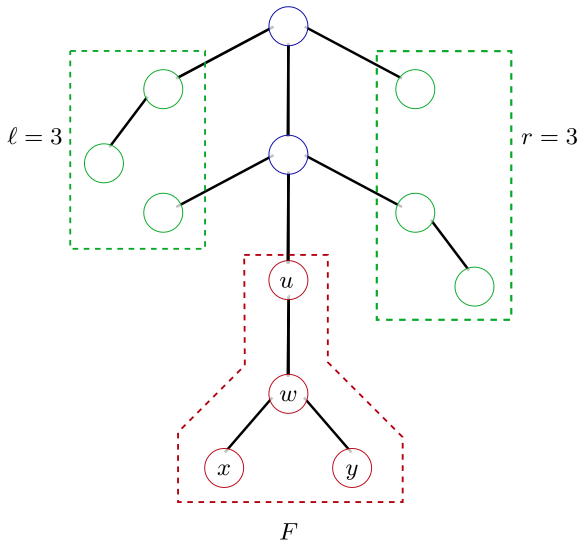

Figure 6 An example of the multiple solutions case, where $l = 3, r = 3$. Before executing step 3 of the algorithm, the remaining F consists of $\{u, w, x, y\}$. Then the algorithm returns three possible solutions: $\{u, w, x, y\}, \{w, x, y\}$, and $\{x, y\}$.

We first consider the case where, among all possible node choices for v, there are two such that neither is an ancestor of the other. Then pick the leftmost (with respect to post-order traversal) and the rightmost (with respect to preorder traversal) of such possibilities for v, denoted v_1 and v_2 respectively. Let G_1 and G_2 be the subtrees in T rooted at v_1 and v_2, respectively. Then by the assumptions on F we necessarily have

$$|\mathsf{size}(RU_{G_1}) - b|, |\mathsf{size}(RU_{G_2}) - b| \le k.$$

Note that $RU_{G_2} \subseteq RU_{G_1}$. Write $D = RU_{G_1} \setminus RU_{G_2}$ for the difference of the right upper part of G_1 and the right upper part G_2. Thus, by triangle inequality, we get that

$$\mathsf{size}(D) = \mathsf{size}(RU_{G_1}) - \mathsf{size}(RU_{G_2}) \le 2k.$$

By our choice of v_1 and v_2, we know that any possible choice for v is either a node in D or an ancestor of v_1. For the former case, we have already shown that there are at most $O(k)$ nodes in D. In the latter case, each distinct v which is an ancestor of v_1 determines a subforest F of a different size. Then because we are assuming that $|\mathsf{size}(F) - c| \le k$, there are only $O(k)$ possibilities for the choices of v which are ancestors of v_1.

The previous argument applies whenever there are two choices for v, neither of which is an ancestor of the other. If there do not exist such options for v, then all possible choices of v lie on the a single root-to-leaf path of T. By the same reasoning as before, the number of possible cases for v here is again at most $O(k)$, because each v would determine a different-sized subforest and $\mathsf{size}(F)$ is allowed to take on $O(k)$ distinct values.

This completes the proof of Lemma 13. As noted earlier, this implies Lemma 9. We conclude by tying these results back to our main theorem.

Proof of Theorem 1. Set up a table which can be indexed by pairs of subforests (F_1, F_2) of T_1 and T_2. Begin using Klein's dynamic programming approach outlined in Section 3 and Lemma 6 but avoid generating subproblems according to the pruning rules described in Proposition 7 and Proposition 8, and store solutions $\texttt{ed}(F_1, F_2)$ produced. In particular, when Klein's algorithm would normally generate a subproblem, we first check if the produced subproblem would be a useful state according to our previous definitions. Proposition 8 and the proof of Lemma 13 make it clear that we can quickly check if a state is useful provided we know the sizes of $F_1, F_2, LU_{F_1}, RU_{F_1}, LU_{F_2}$, and RU_{F_2}, and this information can be kept track of easily simply by updating the sizes according to the type of transition we follow in the table.

So, we can compute $\texttt{ed}(T_1, T_2)$ while only computing $\texttt{ed}(F_1, F_2)$ for useful states. By Lemma 6 there are $O(n \log n)$ possibilities for F_1 and by Lemma 9 there are $O(k^2)$ choices for F_2 for each F_1. So overall we only fill in at most $O(nk^2 \log n)$ entries of the DP table. Since we do a constant amount of work to get the value at each entry of the table, our algorithm has the desired running time. ◀

5 Open problems

For trees of bounded edit distance $k = O(1)$ our algorithm runs in linear time. However, for larger tree edit distances $k = \Theta(n)$ our algorithm requires $O(n^3 \log n)$ time, which is slower than the fastest known algorithm [17] for general tree edit distance by a logarithmic factor. This motivates the question: can we solve the bounded tree edit distance problem in $O(nk^2)$ time instead of $O(nk^2 \log n)$?

The easier problem of *string* edit distance can be solved in $\widetilde{O}(n + k^2)$ time [30, 28], which is quasilinear even for super constant distance parameter $k = O(\sqrt{n})$. This motivates the question of whether it is possible to get similar speedups for tree edit distance. It would be especially interesting to see if the bounded tree edit distance problem can be solved in $\widetilde{O}(n + k^3)$ time. Perhaps the suffix tree techniques used in [33] (and discussed in [11, Appendix]) could prove useful in showing such a result.

Regarding variants of tree edit distance, it remains an open question to get faster algorithms for the harder problem of *unrooted* tree edit distance [24, 18] (where the elementary operations are edge contraction, insertion, and relabeling) when the distance is bounded by k. The best known algorithm for unrooted tree edit distance was recently given by Dudek and Gawrychowski [18] and runs in $O(n^3)$ time. The previous $O(n^3 \log n)$ time algorithm by Klein [24] also applies to the unrooted setting. Although we extended Klein's algorithm to tackle the rooted tree edit distance problem in $O(nk^2 \log n)$ time, it is not obvious how to extend their approach to the unrooted bounded distance setting. This is because Klein solves the unrooted version of the problem by dynamic programming over the subproblems generated by all possible rootings of T_2. This is fine for computing general edit distance because the number of subforests over all possible rootings is $O(n^2)$ just like the number of subforests for a fixed rooted tree on n nodes. However, when the tree edit distance is bounded, the number of possible relevant subproblems over all possible rootings can be $\Omega(n)$ even when k is small. Although our algorithm can be used to recover a near quadratic time algorithm for unrooted tree edit distance when $k = O(1)$ is constant, it remains open whether we can obtain a quasilinear time algorithm in this setting.

Finally, although general tree edit distance with *arbitrary weights* cannot be solved in truly subcubic time unless certain popular conjectures are false [13], analogous fine-grained hardness results rule out truly subquadratic time algorithms for string edit distance even when deletions and insertions have unit cost [6]. Can we show conditional hardness for tree edit distance with unit costs, or can we find a subcubic time algorithm for this problem?

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *Proceedings of the 56th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 59–78, 2015. doi:10.1109/FOCS.2015.14.

2 Tatsuya Akutsu, Daiji Fukagawa, and Atsuhiro Takasu. Approximating tree edit distance through string edit distance. *Algorithmica*, 57(2):325–348, 2010. doi:10.1007/s00453-008-9213-z.

3 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 377–386, 2010. doi:10.1109/FOCS.2010.43.

4 Alexandr Andoni and Negev Shekel Nosatzki. Edit distance in near-linear time: it's a constant factor. In *Proceedings of the 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2020. arXiv:2005.07678.

5 Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. *SIAM J. Comput.*, 41(6):1635–1648, 2012. doi:10.1137/090767182.

6 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

7 John Bellando and Ravi Kothari. Region-based modeling and tree edit distance as a basis for gesture recognition. In *Proceedings of the 10th International Conference on Image Analysis and Processing (ICIAP)*, pages 698–703. IEEE Computer Society, 1999. doi:10.1109/ICIAP.1999.797676.

8 Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005. doi:10.1016/j.tcs.2004.12.030.

9 Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008. doi:10.1016/j.tcs.2008.08.042.

10 Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and MapReduce. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1170–1189, 2018. doi:10.1137/1.9781611975031.76.

11 Mahdi Boroujeni, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, and Saeed Seddighin. $1+\varepsilon$ approximation of tree edit distance in quadratic time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 709–720, 2019. doi:10.1145/3313276.3316388.

12 Joshua Brakensiek and Aviad Rubinstein. Constant-factor approximation of near-linear edit distance in near-linear time. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 685–698, 2020. doi:10.1145/3357713.3384282.

13 Karl Bringmann, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Tree edit distance cannot be computed in strongly subcubic time (unless APSP can). *ACM Trans. Algorithms*, 16(4):48:1–48:22, 2020. doi:10.1145/3381878.

14 Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 141–152, 2003. doi:10.1016/B978-012722442-8/50021-5.

15 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *Proceedings of the 59th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 979–990. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00096.

16 Sudarshan S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 90–101, 1999. URL: http://www.vldb.org/conf/1999/P8.pdf.

17 Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms*, 6(1):2:1–2:19, 2009. doi:10.1145/1644015.1644017.

18 Bartłomiej Dudek and Paweł Gawrychowski. Edit distance between unrooted trees in cubic time. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 45:1–45:14, 2018. doi:10.4230/LIPIcs.ICALP.2018.45.

19 Serge Dulucq and Hélène Touzet. Analysis of tree edit distance algorithms. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 83–95. Springer, 2003. doi:10.1007/3-540-44888-8_7.

20 Serge Dulucq and Hélène Touzet. Decomposition algorithms for the tree edit distance problem. *J. Discrete Algorithms*, 3(2-4):448–471, 2005. doi:10.1016/j.jda.2004.08.018.

21 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33, 2009. doi:10.1145/1613676.1613680.

22 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

23 Matthias Höchsmann, Thomas Töller, Robert Giegerich, and Stefan Kurtz. Local similarity in RNA secondary structures. In *Proceedings of 2nd IEEE Computer Society Bioinformatics Conference, CSB*, pages 159–168. IEEE Computer Society, 2003. doi:10.1109/CSB.2003.1227315.

24 Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA)*, volume 1461 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 1998. doi:10.1007/3-540-68530-8_8.

25 Philip N. Klein, Thomas B. Sebastian, and Benjamin B. Kimia. Shape matching using edit-distance: an implementation. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms (SODA)*, pages 781–790, 2001. URL: http://dl.acm.org/citation.cfm?id=365411.365779.

26 Philip N. Klein, Srikanta Tirthapura, Daniel Sharvit, and Benjamin B. Kimia. A tree-edit-distance algorithm for comparing simple, closed shapes. In David B. Shmoys, editor, *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 696–704, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338628.

27 Michal Koucký and Michael E. Saks. Constant factor approximations to edit distance on far input pairs in nearly linear time. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 699–712. ACM, 2020. doi:10.1145/3357713.3384307.

28 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

29 William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.

30 Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986. doi:10.1007/BF01840446.

31 Thomas B. Sebastian, Philip N. Klein, and Benjamin B. Kimia. Recognition of shapes by editing their shock graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(5):550–571, 2004. doi:10.1109/TPAMI.2004.1273924.

32 Bruce A. Shapiro and Kaizhong Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Bioinformatics*, 6(4):309–318, October 1990. doi:10.1093/bioinformatics/6.4.309.

33 Dennis E. Shasha and Kaizhong Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms*, 11(4):581–621, 1990. doi:10.1016/0196-6774(90)90011-3.

34 Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979. doi:10.1145/322139.322143.

35 Hélène Touzet. A linear tree edit distance algorithm for similar ordered trees. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3537 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 2005. doi:10.1007/11496656_29.

36 Hélène Touzet. Comparing similar ordered trees in linear-time. *J. Discrete Algorithms*, 5(4):696–705, 2007. doi:10.1016/j.jda.2006.07.002.

37 Esko Ukkonen. Algorithms for approximate string matching. *Inf. Control.*, 64(1-3):100–118, 1985. doi:10.1016/S0019-9958(85)80046-2.

38 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.

39 Michael S. Waterman. *Introduction to computational biology: maps, sequences and genomes.* CRC Press, 1995.

40 Kaizhong Zhang and Dennis E. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989. doi:10.1137/0218082.