$1+\epsilon$ Approximation of Tree Edit Distance in Quadratic Time*†

Mahdi Boroujeni
Sharif University of Technology
Tehran, Iran
safarnejad@ce.sharif.edu

Mohammad Ghodsi
Sharif University of Technology
Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
ghodsi@sharif.edu

MohammadTaghi Hajiaghayi‡
University of Maryland
College Park, MD, USA
hajiagha@cs.umd.edu

Saeed Seddighin‡
University of Maryland
College Park, MD, USA
sseddigh@umd.edu

ABSTRACT

Edit distance is one of the most fundamental problems in computer science. Tree edit distance is a natural generalization of edit distance to ordered rooted trees. Such a generalization extends the applications of edit distance to areas such as computational biology, structured data analysis (e.g., XML), image analysis, and compiler optimization. Perhaps the most notable application of tree edit distance is in the analysis of RNA molecules in computational biology where the secondary structure of RNA is typically represented as a rooted tree.

The best-known solution for tree edit distance runs in cubic time. Recently, Bringmann *et al.* show that an $O(n^{2.99})$ algorithm for weighted tree edit distance is unlikely by proving a conditional lower bound on the computational complexity of tree edit distance. This shows a substantial gap between the computational complexity of tree edit distance and that of edit distance for which a simple dynamic program solves the problem in quadratic time.

In this work, we give the first non-trivial approximation algorithms for tree edit distance. Our main result is a quadratic time approximation scheme for tree edit distance that approximates the solution within a factor of $1+\epsilon$ for any constant $\epsilon > 0$.

CCS CONCEPTS

• **Theory of computation → Approximation algorithms analysis**; *Graph algorithms analysis*; Data structures design and analysis; Streaming, sublinear and near linear time algorithms; Dynamic programming.

*A portion of this work was completed while some of the authors were visiting Simons Institute for Theory of Computing.

†The omitted proofs can be found in the full version of this paper, which may be found at http://safarnejad.ir/papers/ted-fullversion.pdf

‡Supported in part by NSF CAREER award CCF-1053605, NSF BIGDATA grant IIS-1546108, NSF AF:Medium grant CCF-1161365, DARPA GRAPHS/AFOSR grant FA9550-12-1-0423, and another DARPA SIMPLEX grant.

STOC '19, June 23–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6705-9/19/06…$15.00
https://doi.org/10.1145/3313276.3316388

KEYWORDS

approximation algorithms, graph algorithms, fine-grained complexity, randomized algorithms

ACM Reference Format:
Mahdi Boroujeni, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, and Saeed Seddighin. 2019. $1+\epsilon$ Approximation of Tree Edit Distance in Quadratic Time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on the Theory of Computing (STOC '19), June 23–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3313276.3316388

1 INTRODUCTION

Edit distance is one of the most fundamental problems in combinatorial optimization. It has been subject to many studies since the 60's and even after 50 years, some of the questions regarding its computational complexity are still open. Two natural generalizations of edit distance are tree edit distance and language edit distance. While the known algorithmic results for tree edit distance have been mostly basic and unexciting, recent developments have been very fruitful for language edit distance [12, 25, 30]. In this work, our focus is on approximation algorithms for tree edit distance and present the first non-trivial results for this problem.

Tree edit distance was first introduced by Selkow [31] in the late 70's. Since then, tree edit distance has found its applications in various areas such as computational biology [9, 21, 32, 38], structured data analysis (e.g., XML) [13, 16, 18], image analysis [14], and compiler optimization [17]. Perhaps the most notable application of tree edit distance is in the analysis of RNA molecules in computational biology where the secondary structure of RNA is typically represented as a rooted tree [21, 23].

While in edit distance, the goal is to transform a string s into another string $\bar{s}$, in tree edit distance the goal is to transform a rooted tree T into another rooted tree $\overline{T}$ using the least number of edit operations. We assume that both trees T and $\overline{T}$ are rooted, and there is a left-to-right order between the sibling nodes. Moreover, every node has a label which identifies the type of the node. The elementary operations are *node deletion*, *node addition*, and *node relabel*. In node deletion, we remove a node r and replace it with all of its children, preserving their order. The reverse of node deletion is node addition which allows us to select a consecutive set of siblings and bring them under a new node r which appears at the previous position of the relocated nodes. In node relabel, we simply modify the label of an existing node.

The computational aspect of the problem is also widely studied. Tai [36] gives the first solution for tree edit distance that runs in time $O(n^6)$ where n is the total number of nodes in both trees. This was later improved in a series of works to an $O(n^4)$ algorithm [39], and an $O(n^3 \log n)$ algorithm [27]. Finally, Demaine *et al.* provide an $O(n^3)$ time algorithm [17]. Very recently, the seminal work of Bringmann *et al.* [11] proves that the cubic running time barrier for weighted tree edit distance cannot be beaten unless APSP[1] admits a truly subcubic time solution and weighted k-clique[2] admits an $O(n^{k-\epsilon})$ time solution. The existence of such a lower bound was previously conjectured by Abboud [1] in a set of seven open problems. Also, an $O(n\mathsf{d}_{\max}{}^3)$ algorithm is proposed by Touzet [37] that is subcubic when the distance between the two trees is small ($\mathsf{d}_{\max}$ here denotes an upper bound on the solution size). Despite these studies, the literature on tree edit distance is quite poor concerning approximation solutions. The only relevant results are the $O(n^{3/4})$ and $O(\mathsf{h}_{\max})$ approximation algorithms of Akutsu *et al.* [2] for degree-bounded trees that run in time $O(n^2)$. $\mathsf{h}_{\max}$ here denotes an upper bound on the heights of the two trees. A quadratic time algorithm with approximation factor $O(n^{2/3})$ follows from the algorithm of Touzet [37] by solving the problem for instances whose distance is smaller than $n^{1/3}$ and reporting a solution of $O(n)$ for instances with a distance of at least $n^{1/3}$. In contrast to tree edit distance, approximation algorithms for edit distance have been subject to many studies [4, 5, 7, 8, 22, 24, 28], culminating in a poly(log) approximation algorithm in linear time. Recently, a quantum algorithm is given for edit distance that approximates the solution within a constant factor in truly subquadratic time [10] by exploiting triangle inequality. Subsequent work discovers a novel classic replacement for the quantum techniques and obtains a truly subquadratic time algorithm within a constant factor for classic computers [15].

In this work, we present a $1+\epsilon$ approximation algorithm for tree edit distance that runs in time $\widetilde{O}(n^2)$. We show that the running time of our algorithm improves to $\widetilde{O}(n\mathsf{d}_{\max})$ when the solution size is guaranteed to be bounded by $\mathsf{d}_{\max}$. Our results also imply an almost linear time algorithm ($\widetilde{O}(n)$) with an approximation factor of $O(\sqrt{n})$. Although the recent result of [11] suggests that weighted tree edit distance is strictly (computationally) harder than edit distance, our results suggest that both problems may be equally time-consuming, concerning approximation algorithms for the unweighted case. Table 1 compares our results to the previously known solutions. Tree edit distance is a generalization of edit distance; therefore, a $1+\epsilon$ approximation algorithm for tree edit distance is hard to acheive unless edit distance admits a truly subquadratic $1+\epsilon$ approximation scheme.

We obtain our result through several combinatorial ideas. Some of these ideas such as heavy-light decomposition, or reducing the problem to forest edit distance (see Section 2 for a definition) have been used in the previous work [17, 27, 37]. These techniques are very classic as almost any algorithm for TED uses these ideas. However, the main techniques that enable us to achieve a $1+\epsilon$ approximate solution are pretty novel and to the best of our knowledge have not been used in previous work.

Table 1: In the bounded TED problem, we are guaranteed that the distance between the two trees is bounded by $\mathsf{d}_{\max}$. $\mathsf{h}_{\max}$ in the algorithm of [2] denotes an upper bound on the heights of the trees. ⋆ follows from the $O(n\mathsf{d}_{\max}{}^3)$ algorithm of Touzet [37].

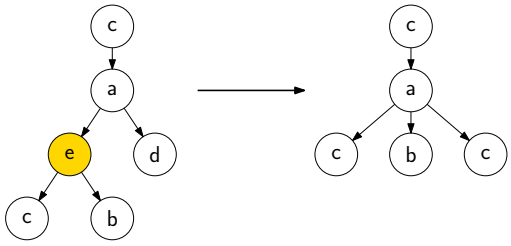
Our Results			
Problem	Reference	Approximation Ratio	Running Time
TED	Theorem 5.3	$1+\epsilon$	$\widetilde{O}(n^2)$
bounded TED	Full version	$1+\epsilon$	$\widetilde{O}(n\mathsf{d}_{\max})$
TED	Full version	$O(\sqrt{n})$	$\widetilde{O}(n)$
Previous Work			
TED	[36]	exact	$O(n^6)$
TED	[39]	exact	$O(n^4)$
TED	[27]	exact	$O(n^3 \log n)$
TED	[17]	exact	$O(n^3)$
bounded TED	[37]	exact	$O(n\mathsf{d}_{\max}{}^3)$
TED	[2]	$O(n^{3/4})$	$O(n^2)$
TED	[37]⋆	$O(n^{2/3})$	$O(n^2)$
TED	[2]	$O(\mathsf{h}_{\max})$	$O(n^2)$

2 PRELIMINARIES

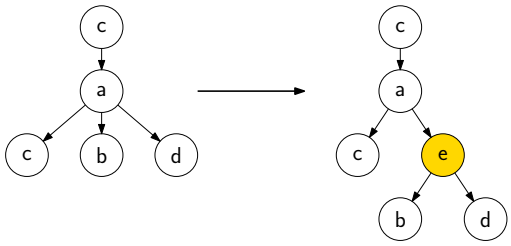
Given two ordered rooted trees T and $\overline{T}$, the tree edit distance problem (TED) seeks to transform one of the trees into another one via the minimum number of operations. In TED, we assume both trees are rooted, each node has a label, and the children of each node are ordered. We call two subtrees *identical* if the roots' labels are the same, the number of the children of the roots are equal, and the subtrees of the children of the roots are also identical in the same order. In each operation, we are allowed to remove a node, add a new node, or relabel an existing node, all at the same cost of one. In each case, the order of the siblings remains the same in each neighborhood. When we remove a node, it will be replaced by its children (if any) without any change in the order of its siblings or that of its children. Similarly, when we wish to add a new node, we are allowed to select a number of consecutive siblings (in an arbitrary neighborhood) and add the new node as their father. In this case, the newly added node takes their place, and the replaced nodes appear as its children in the same order. It can also be the case that we simply add a new node without any children at any position in a tree. Figure 1 illustrates how node deletion, node addition, and node relabel modify a tree.

To simplify the notation, we represent each tree with a *balanced* string of parentheses. Balanced here implies that every opening parenthesis has a corresponding closing parenthesis and that the pairs of parentheses are correctly nested. In this representation, each node u is represented with a pair of opening and closing parentheses which enclose the children of u in the order they appear in

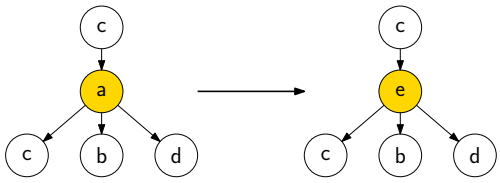
[1] finding all pairs shortest paths in a graph.
[2] In the weighted k-clique problem, we are given an undirected weighted graph on n nodes, and $O(n^2)$ edges with integral weights, and we seek to find a k-clique with the highest total sum of edge weights.



(a) The node with label e is deleted.



(b) The node with label e is inserted.



(c) The node with label a is relabeled to e.

Figure 1: Three basic operations to transform a tree into another one. Modified nodes are highlighted in each example.

the tree. This representation always corresponds to a sequence of trees which we call an *ordered forest*. When the first and the last characters of the representation match, it means that the representation corresponds to a tree, and all of the nodes are nested under a single node which is the root of the tree.

Every pair of matching parentheses has a label identical to that of its corresponding node in the tree. We show this label on top of the parentheses. For example, a leaf with a label a is shown by "$\overset{aa}{()}$". This way, node deletion, node addition, and node relabel are equivalent to deleting a matching pair of parentheses, adding a matching pair of parentheses, or relabeling an existing pair of matching parentheses. When we add a node, we select a number of consecutive siblings to be its children. Similarly, when we add a new pair of parentheses, we select a substring, where the match of every parenthesis is inside the substring. Figure 2 shows how these operations change the representations of the trees.

For a string s ($\bar{s}$) which represents a tree or an ordered forest, we denote its i'th character by $s[i]$ ($\bar{s}[i]$). We also denote a substring of s ($\bar{s}$) from the ℓ'th character to the r'th character by $s[\ell, r]$ ($\bar{s}[\ell, r]$). For a substring $s[\ell, r]$, we call a character $s[i]$ *redundant*, if the matching parenthesis of $s[i]$ is outside of $s[\ell, r]$. Based on this, the *refined* subsequence of a substring $s[\ell, r]$ is the sequence of all characters in $s[\ell, r]$ that are not redundant. For a tree or an ordered forest represented by s, we call the refined subsequence of

caeccbbeddac $\rightarrow$ caccbbddac

$(((\underline{(}()())\underline{)}())) \rightarrow ((()()()))$

(a) The node with label e is deleted.

caccbbddac $\rightarrow$ caccebbddeac

$((()()())) \rightarrow (((\underline{(}()()\underline{)}\underline{)}))$

(b) The node with label e is inserted.

caccbbddac $\rightarrow$ ceccbbddec

$(((\underline{(}()()())\underline{)})) \rightarrow (((\underline{(}()()())\underline{)}))$

(c) The node with label a is relabeled to e.

Figure 2: Three basic operations to transform a tree into another one. The representations belong to trees of Figure 1. Modified parentheses are underlined in each example.

a substring $s[\ell, r]$ a *subforest* of s. Moreover, a subforest is *proper* if it is the refined subsequence of a substring that has no redundant closing parenthesis. Throughout this paper, we use T when we refer to a forest or a tree, and we use s when we refer to its string representation. We also use $\overline{T}$ and $\bar{s}$ when we refer to a second forest or tree, or its string representation. Moreover, we denote the nodes of a tree with characters u and v and use x and y for their corresponding parentheses in the string representations. We also denote the labels of the nodes with characters a, b, c, d, and e. Similar to TED, forest edit distance (FED) seeks to transform one ordered forest into another one with the minimum number of operations.

For two ordered rooted trees T and $\overline{T}$, we denote their tree edit distance by $\text{ted}(T, \overline{T})$. Similarly, for the string representations of two ordered rooted trees s and $\bar{s}$ we denote it by $\text{ted}(s, \bar{s})$. Moreover, we denote the forest edit distance between two ordered forests s and $\bar{s}$ by $\text{fed}(s, \bar{s})$. For two nodes u and $\bar{u}$ in T and $\overline{T}$, respectively, we define $\text{ted}(u, \bar{u})$ as the tree edit distance between subtrees rooted by u and $\bar{u}$ with an additional assumption that node u is mapped to node $\bar{u}$. Note that $\text{ted}(r, \bar{r})$ may differ from $\text{ted}(T, \overline{T})$, where r and $\bar{r}$ are roots of T and $\overline{T}$, respectively, because of the additional assumption. In our algorithm, we add two dummy roots in the beginning to the two trees, compute the ted of the two new roots and output it as the ted between two input trees.

In addition, we denote the edit distance between two strings s and $\bar{s}$ by $\text{ed}(s, \bar{s})$. The size of the subtree induced by a node u is shown by $\text{size}(u)$. Moreover, let isClosing (isOpening) be a function that gets a parenthesis as input and outputs 1 (0) if it is a closing parenthesis and 0 (1) otherwise. Note that $\widetilde{O}$ is similar to big O notation but ignores $\text{poly}(\log n)$ and $\text{poly}(1/\epsilon)$ factors.

3 OUR RESULTS AND TECHNIQUES

Let us begin by explaining a naïve $O(n^4)$ dynamic programming algorithm for tree edit distance. We refer to the input trees by T and $\overline{T}$ and denote the corresponding representations of the given trees by s and $\bar{s}$, respectively. Since both trees have n nodes in total, the number of pairs of matching parentheses in s and $\bar{s}$ sum to n. Notice that, each pair of matching parentheses in a string denotes a subtree. Thus, we define the tree edit distance between two pairs of parentheses of s and $\bar{s}$ as the tree edit distance between their

corresponding subtrees. In this algorithm, we compute the tree edit distance between every pair of parentheses x and y of s and $\bar{s}$ in a bottom-up order. That is, we start with the leaves of the trees and move on to their parents until we compute the solution for the roots of the two trees.

Let x and $\bar{x}$ be two matching pairs of parentheses of s and $\bar{s}$ that correspond to nodes u and $\bar{u}$ of the trees. Since we solve the problem in a bottom-up order, when we wish to compute the tree edit distance between x and $\bar{x}$, the solution is given for every pair of the children of x and $\bar{x}$. Thus, the problem that we need to solve is the following:

Let t and $\bar{t}$ be two balanced strings of parentheses corresponding to two forests F and $\bar{F}$. Along with t and $\bar{t}$, we are given the tree edit distance of every pair of matching parentheses of the two strings. The goal is to compute the smallest number of edit operations on these strings to transform t into $\bar{t}$.

The operations that we are allowed to perform are the tree edit operations described in Section 2. We call this problem *forest edit distance* (FED). A similar definition of this problem is also given in [39]. Notice that a fundamental difference between tree edit distance and forest edit distance is that in forest edit distance, all the tree edit distances are given in the input whereas, in tree edit distance, the input only contains the two trees. We elaborate on forest edit distance later in this section.

The naïve $O(n^4)$ time algorithm for tree edit distance uses forest edit distance as a black box to find the tree edit distances between every pair of nodes of the trees. It has been shown that forest edit distance can be solved in time $O(n^2)$ for two forests with n nodes in total [39]. Since we use FED for every pair of nodes of the two trees, the total running time of our algorithm is $O(n^4)$. Indeed by fixing a parameter $\mathsf{h}_{\max}$ to be an upper bound on the height of the two trees, one can show that the same algorithm runs in time $O(n^3\mathsf{h}_{\max})$ which is $\widetilde{O}(n^3)$ for balanced trees. The classic tree decomposition of Sleator and Tarjan [34] (called *heavy-light tree decomposition*) is then used by Klein [27] to improve the running time to $\widetilde{O}(n^3)$. Roughly speaking, Sleator and Tarjan show that any tree can be decomposed into a number of *spines* such that in any path from the root to any leaf of the tree, we cross at most $O(\log n)$ spines. Moreover, such a decomposition can be found in linear time. Thus, an algorithm for solving the problem for two spines leads to a solution for the whole trees with a logarithmic overhead. This technique has been applied to a variety of algorithms [6, 17, 27, 35] to break the linear dependence on the height of the trees.

In order to design a $1+\epsilon$ approximation algorithm, we too make use of the heavy-light tree decomposition of Sleator and Tarjan [34]. Based on this decomposition and the analysis that we present in Section 5.3, an $\widetilde{O}(n^2)$ time algorithm for approximating TED follows from a similar algorithm that solves the problem for two spines of the trees with similar running time and approximation factor (see Figure 3). To be more specific, let us clarify what we mean by solving the problem for two spines of the trees. In the heavy-light decomposition, the vertices are decomposed into a set of disjoint spines. Every spine has a property that the depth of the vertices increase as we traverse the spine. Therefore, the second node of a spine is a child of the first node; the third node is a child of the second node and so on.

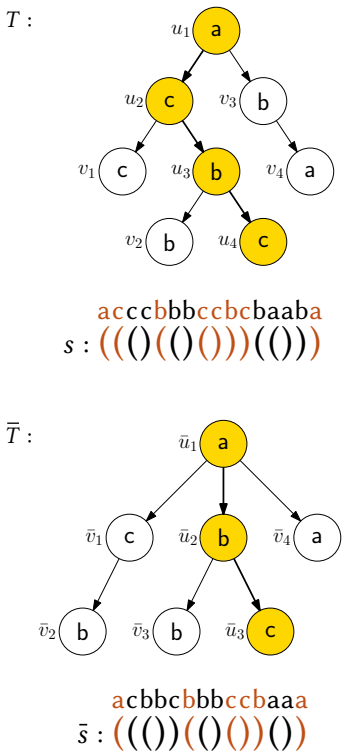


Figure 3: A spine for each of trees T and $\bar{T}$ is illustrated in this example. Spine nodes are highlighted. The string representations of the two trees s and $\bar{s}$ are shown below each tree. The parentheses are colored accordingly.

The naïve $O(n^4)$ algorithm constructs the solution in a bottom-up manner and for every pair of vertices, uses FED to solve the problem. When the spines are involved, we do the same thing except that we compute the solution for all vertices of a spine in a single shot. Therefore, the depth of the recursion reduces to $O(\log n)$. Now, assume that we are given two spines $S = (u_1, u_2, \ldots)$ and $\bar{S} = (\bar{u}_1, \bar{u}_2, \ldots)$ of the two trees and we would like to solve the problem for the subtrees rooted under u_i and $\bar{u}_j$ for the vertices of the spines (see Figure 4 for the inputs and outputs of the problem). That is, for any pair of nodes u_i and $\bar{u}_j$, we would like to compute/approximate $\mathsf{ted}(u_i, \bar{u}_j)$. However, additional important information is also provided: for any two vertices v and $\bar{v}$ such that either $v \notin S$ or $\bar{v} \notin \bar{S}$, $\mathsf{ted}(v, \bar{v})$ is given. This information is available since we solve the problem for the spines in a bottom-up order. We call this problem *spine edit distance*. Spine edit distance generalizes the FED problem since if both spines have only one node, the resulting problem would be equivalent to FED for the children of the two nodes. Figure 4 shows the input and output of the problem for two given spines.

Indeed, our main challenge is to approximate the solution for two spines. In an instance of spine edit distance, let us call the nodes that appear in the spines ($u_1, u_2, \ldots$ and $\bar{u}_1, \bar{u}_2, \ldots$) the highlighted nodes and similarly, we call the parentheses corresponding to these nodes highlighted parentheses. We refer to the rest of the nodes and

ted	$\bar{u}_1$	$\bar{u}_2$	$\bar{u}_3$	$\bar{v}_1$	$\bar{v}_2$	$\bar{v}_3$	$\bar{v}_4$
u_1				6	3	2	1
u_2				7	4	2	1
u_3				7	4	2	1
u_4				7	5	3	1
v_1	6	2	0	1	1	1	1
v_2	6	2	1	1	0	0	1
v_3	6	2	2	2	1	1	1
v_4	6	3	1	2	1	1	0

Input

ted	$\bar{u}_1$	$\bar{u}_2$	$\bar{u}_3$
u_1	3	5	7
u_2	3	2	4
u_3	4	0	2
u_4	6	2	0

Output

Figure 4: The inputs and outputs of an instance of the spine edit distance problem corresponding to the example of Figure 3. In spine edit distance, the input contains the tree edit distances between all pairs of nodes except the pairs that are both highlighted. The input also contains two trees each with a spine. In the output, we should compute/approximate the solution for the pairs of highlighted nodes.

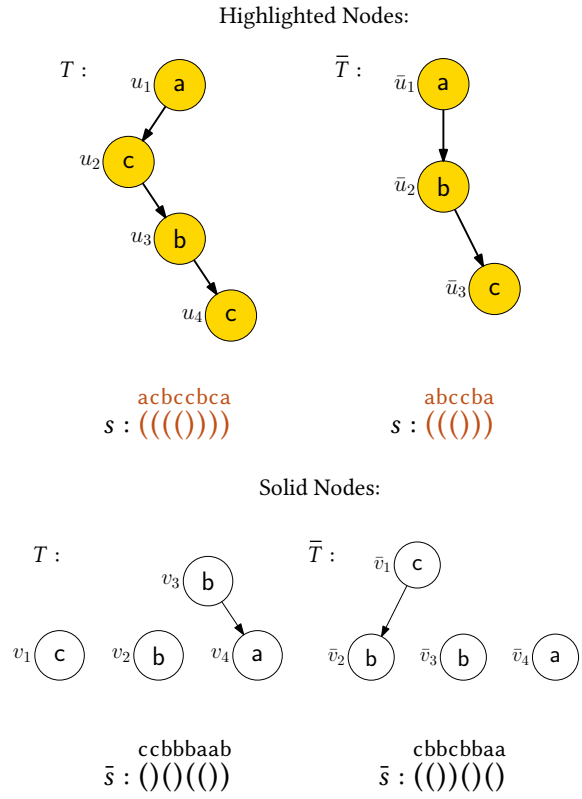


Figure 5: The example of Figure 3 is separated into an instance of edit distance and an instance of forest edit distance.

parentheses in the subtrees as solid nodes and solid parentheses. Spine edit distance seeks to find a solution for two intertwined FED and ED problems (see Figure 5). On the one hand, if we only take into account the highlighted nodes and ignore the costs for the rest of the nodes, the problem becomes an instance of ED. On the other hand, if we ignore these nodes and only consider the solid nodes, the problem becomes an instance of FED. Both ED and FED admit $O(n^2)$ time solutions even in weighted cases; however, there is a conditional lower bound of $\Omega(n^{3-o(1)})$ on the computational complexity of weighted spine edit distance due to [11][3].

In Section 5.2, we explain our $1+\epsilon$ approximation algorithm for spine edit distance which also carries over to TED. The intuition behind our algorithm is the following: Recall that the goal of the spine edit distance problem is to find the tree edit distance between the nodes of the two spines. Fix a node $u_i \in S$ and a node $\bar{u}_j \in \bar{S}$ and assume that the goal is to approximate $\mathrm{ted}(u_i, \bar{u}_j)$ subject to node u_i being transformed into node $\bar{u}_j$. Assume that $\mathrm{ted}(u_k, \bar{u}_l)$ is known for every $k > i$ and $l > j$. Moreover, assume that we are given a highlighted node $u_{i'}$ and a node w and are guaranteed that there exists an optimal way to transform the subtree rooted by u_i to the subtree rooted by $\bar{u}_j$ in a way that node $u_{i'}$ transforms into w and that all highlighted nodes in between u_i and $u_{i'}$ are removed and all nodes in the path from $\bar{u}_j$ to w are inserted. Note that w can be either a highlighted or a solid node in the subtree of $\bar{u}_j$. Provided that this information is correct, we can then formulate $\mathrm{ted}(u_i, \bar{u}_j) := \mathrm{ted}(u_{i'}, w) + R + C + L$ as the solution corresponding to the nodes u_i and $\bar{u}_j$. In the above formulation, $R := (i' - i - 1) + (\mathrm{depth}(w) - j - 1)$ denotes the cost of removing the nodes between u_i and $u_{i'}$ and inserting the nodes between $\bar{u}_j$ and w. Moreover, C denotes the cost of transforming the solid nodes in between u_i and $u_{i'}$ (solid nodes in the subtree rooted by u_i but not in the subtree rooted by $u_{i'}$) to the solid nodes in between $\bar{u}_j$ and w. In addition, L denotes the cost of changing the label of u_i to the label of $\bar{u}_j$. Indeed, R and L can be computed in time $O(1)$; however, computing C may be time-consuming. As we show in Section 5.2, the problem of computing C essentially reduces to solving FED.

The first step of our algorithm is constructing a data structure that enables us to approximate C in the above formulation in time $\widetilde{O}(1)$. We call this data structure FEDDS. Provided that FEDDS is available, we can approximate $\mathrm{ted}(u_i, \bar{u}_j) := \mathrm{ted}(u_{i'}, w) + R + C + L$ in time $\widetilde{O}(1)$. Since the highlighted nodes of the first forest are ignored for computing C, we only incorporate the solid nodes of the first forest for determining C. Thus, one can interpret FEDDS as a data structure that receives an instance of FED as input and answers the queries of the following type in time $\widetilde{O}(1)$:

Let t and $\bar{t}$ be the string representations for two forests F and $\bar{F}$. Given two intervals $[\ell, r]$ and $[\bar{\ell}, \bar{r}]$ of the input strings such that both $t[\ell, r]$ and $\bar{t}[\bar{\ell}, \bar{r}]$ are balanced, what is the forest edit distance between $t[\ell, r]$ and $\bar{t}[\bar{\ell}, \bar{r}]$?

Notice that the special case of the above problem is when both $[\ell, r]$ and $[\bar{\ell}, \bar{r}]$ span the entire length of the two string representations and thus in the output, we have to compute the FED of the two

[3] A subcubic time solution for spine edit distance yields a subcubic time solution for TED.

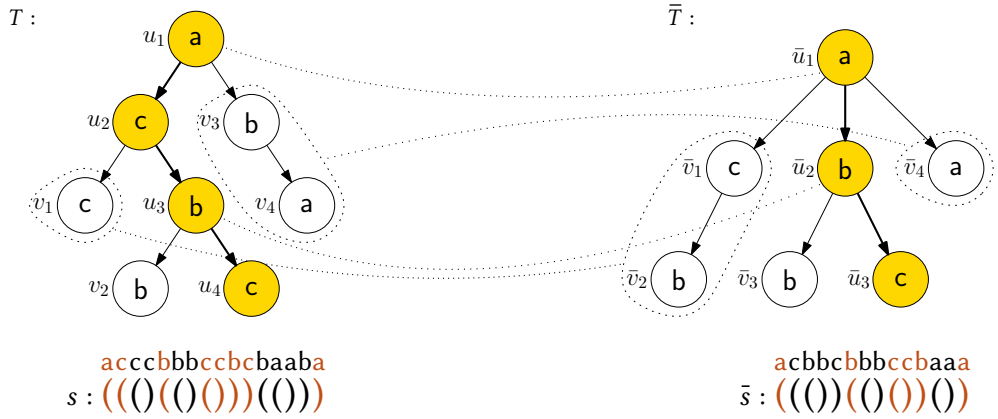


Figure 6: This example illustrates how ted$(u_1, \bar{u}_1)$ is computed from ted$(u_3, \bar{u}_2)$ in the example of Figure 3. ted$(u_1, \bar{u}_1)$ = ted$(u_3, \bar{u}_2) + R + C + L = 0 + 1 + 2 + 0 = 3$.

forests. In Section 4.2, we design an algorithm for FEDDS with preprocessing time $\widetilde{O}(n^2)$ and query time $\widetilde{O}(1)$. Our data structure approximates the solution within a factor of $1+\epsilon$. The high-level idea is that we construct $O(\log_{1+\epsilon} n)$ data structures FEDDS$_k$ where $k = (1+\epsilon)^i$ for $0 \le i \le \log_{1+\epsilon} n$. Each FEDDS$_k$ is responsible for answering the queries whose solutions are close to k. For each FEDDS$_k$, we break the input strings into roughly $O(n/k)$ marked points. After a preprocessing in time $O(n)$, for every pair of marked points, we design an $\widetilde{O}(k^2)$ time algorithm to compute the forest edit distance up to a threshold of $O(k)$ from the beginning of the marked points. Then, we show that since the additive error for FEDDS$_k$ is allowed to be as large as ϵk, we can afford to modify each query to make sure both intervals of each query start from marked points. Hence, using the precomputed information, we can answer each query in time $\widetilde{O}(1)$. This data structure is explained in Section 4.2.

Theorem 4.3 [restated informally]. *For any $\epsilon > 0$,* FEDDS *can be constructed in time $\widetilde{O}(n^2)$. Then it can output* fed *between any two proper subforests in time $\widetilde{O}(1)$ within an approximation factor of $1+\epsilon$.*

Let us get back to the computation of spine edit distance. Now that FEDDS is available, given that for a pair of nodes u_i and $\bar{u}_j$ their solution is derived from $u_{i'}$ and w, one can compute ted$(u_i, \bar{u}_j)$ in time $\widetilde{O}(1)$. However, $u_{i'}$ and w are not known in advance. Indeed one can try $O(n^2)$ possibilities for $u_{i'}$ and w and solve the problem in time $\widetilde{O}(n^2)$ for u_i and $\bar{u}_j$ and in time $\widetilde{O}(n^4)$ for all pairs of nodes of the two spines. However, this running time is not desirable. We improve this algorithm significantly by exploiting the following facts:

- As the distance between u_i and $u_{i'}$ increases, R also increases. Therefore, if $\mathsf{d}_{\max}$ is an upper bound on the solution, then $i' \le i + \mathsf{d}_{\max} + 1$ holds.
- The difference between the number of solid nodes on the right of $u_{i'}$ and the number of solid nodes on the right of w appears in part C of the solution. Therefore, w also has $O(\mathsf{d}_{\max})$ possibilities.
- ted$(u_i, \bar{u}_j) = d$ is a desirable estimate for the tree edit distances of the highlighted pairs of nodes that are no more than ϵd away from u_i and $\bar{u}_j$ in the string representations.

We explain these abstract ideas in details in Section 5.2 and show how this gives us a $1+\epsilon$ approximate solution for spine edit distance and in turn for tree edit distance. We face several challenges to approximate the solution within a factor of $1+\epsilon$. We briefly point out some of the difficulties in the following:

- Our approach for solving spine edit distance is dynamic, and we lose an error of $1+\epsilon$ in every step. We have to make sure the error does not propagate.
- We may have different (additive) error thresholds based on the values of ted$(u_i, \bar{u}_j)$. Our algorithm should be careful about the error thresholds without having prior information about the solution.

The above challenges make our algorithm quiet involved and non-trivial. Finally, it follows that once we get a $1+\epsilon$ approximate solution for spine edit distance, we can turn that into an algorithm for TED with roughly the same running time.

Theorem 5.3 [restated informally]. *For any $\epsilon > 0$, TED admits an $\widetilde{O}(n^2)$ time algorithm with approximation factor $1+\epsilon$.*

In Section 6 of the full version of the paper, we revisit the above ideas for the case where the tree edit distance between the two trees is guaranteed to be at most $\mathsf{d}_{\max}$ and show that the running time of our algorithm improves to $\widetilde{O}(n\mathsf{d}_{\max})$. The general idea is that if the opening (closing) parentheses of two nodes in the input trees differ by more than $2\mathsf{d}_{\max}$, we know that the optimal solution does not map them together. Thus, we do not need to compute tree edit distance between them.

Theorem 6.4 [restated informally]. *For any $\epsilon > 0$, TED admits an $\widetilde{O}(n\mathsf{d}_{\max})$ time algorithm with approximation factor $1+\epsilon$, where $\mathsf{d}_{\max}$ is an upper bound on the size of the solution.*

We further design a linear time algorithm for approximating TED in Section 7 of the full version of the paper, with an approximation factor of $O(\sqrt{n})$. First, notice that all the algorithms we discussed so far have a super-linear running time when the distance between the two trees is not constant. Therefore, in order to give a linear time solution, one should go beyond the above ideas. Let us assume for simplicity that a distance d is given to us as input, and our goal is to either approve that the distance between the two trees is at most d or that the solution is much larger than $20d\sqrt{n}$. Note that it is safe to assume that at least one of the two cases holds. Thus, if $d \geq \sqrt{n}$ the solution is always equal to d and the problem is trivial. Hence, we assume w.l.o.g. that $d < \sqrt{n}$. Moreover, we assume that d is the distance between the two trees, and based on that, try to find a solution. If we fail, we realize that the solution is at least $20d\sqrt{n}$.

To illustrate our techniques, let us assume that the two trees have very simple structures. In the extreme case, we consider both trees to be paths. Divide each of the trees into $\sqrt{n}$ disjoint paths of size $\sqrt{n}$ that span the vertices of the two forests. We denote these paths by $P_1, P_2, \ldots, P_{\sqrt{n}}$ for T and $\overline{P}_1, \overline{P}_2, \ldots, \overline{P}_{\sqrt{n}}$ for $\overline{T}$ (see Figure 7). The key idea that enables us to approximate the solution in near-linear time is the following structural properties:

(1) if $\mathrm{ted}(T, \overline{T}) \leq d$ holds then $\mathrm{ted}(P_i, \overline{P}_i) \leq 8d$ also holds for any $1 \leq i \leq \sqrt{n}$.
(2) $\sum_i \mathrm{ted}(P_i, \overline{P}_i) \geq \mathrm{ted}(T, \overline{T})$.

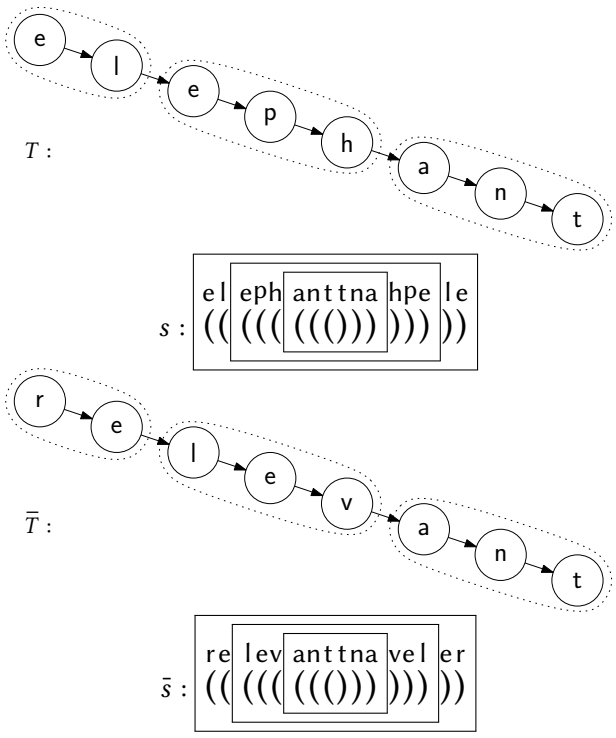


Figure 7: The decomposition of the paths is illustrated in this example

In words, the above two properties show that $\sum_i \mathrm{ted}(P_i, \overline{P}_i)$ gives us an $O(\sqrt{n})$ approximate solution for $\mathrm{ted}(T, \overline{T})$. However, computing $\sum_i \mathrm{ted}(P_i, \overline{P}_i)$ may take up to $O(\sqrt{n^2}) = O(n)$ time for each i and thus if we naïvely find the solution for each $i \in [\sqrt{n}]$, the total running time would be $O(n^{3/2})$.

If the tree edit distance between the two trees is exactly equal to d, we know that both properties (i) and (ii) hold. However, we need to find a way to verify this equality. To this end, we use a randomized procedure and analyze its correctness with concentration bounds. Fix a probability $p = O(\log n/d)$. Select a subset I of the set $\{1, 2, \ldots, \sqrt{n}\}$ such that each number appears in I independently with probability p. Compute $\mathrm{S} = \sum_{i \in \mathrm{I}} \mathrm{ted}(P_i, \overline{P}_i)$ and approximate $\sum_{1 \leq i \leq \sqrt{n}} \mathrm{ted}(P_i, \overline{P}_i)$ with $1/p\mathrm{S}$. We show that if d is an accurate estimate for $\mathrm{ted}(T, \overline{T})$, the approximated value is close to d w.h.p. On the other hand, if $\mathrm{ted}(T, \overline{T})$ is a multiplicative factor $O(\sqrt{n})$ larger than d then the estimate we get in our algorithm is much larger than d w.h.p. and thus we can distinguish the two cases w.h.p. Therefore, this algorithm gives us a correct solution w.h.p. On the computational front, we show that the running time of our algorithm is $\widetilde{O}(n)$ if we use our $\widetilde{O}(n d_{\max})$ time algorithm for estimating the TED's of the paths.

The above ideas lead to an $\widetilde{O}(n)$ time algorithm for approximating the tree edit distance between two paths. To extend this result to general trees, we need a proper decomposition of the trees into smaller components. There already exist several tree decomposition techniques (*e.g.*, separator decomposition based on [26] and microtree/macrotree decomposition of [3]); however, none of these techniques apply to our algorithm. Thus, we introduce a new tree decomposition technique which we call *synchronous decomposition of trees*. For a given $1 \leq \Delta \leq n$, our algorithm decomposes one of the trees into $O(n/\Delta)$ (not necessarily connected) components of size at most $O(\Delta)$. However, our decomposition maintains the property that for each disconnected component, there exists a node in the tree such that adding that node to the component (along with its incident edges) makes the component connected. Our algorithm finds a similar decomposition for the second tree and corresponds the decomposed components together. An example of our synchronous tree decomposition is shown in Figure 8. In our linear time algorithm, we set $\Delta = O(\sqrt{n})$ in the synchronous decomposition.

Via our synchronous decomposition, we are able to apply the above technique to estimate the solution size. However, in contrast to the case of paths, here every decomposed component may be neighbors with $O(\sqrt{n})$ other components. This further complicates the algorithm as $\mathrm{ted}(T_i, \overline{T}_i) \leq 8d$ may not hold for two decomposed components T_i and $\overline{T}_i$ of the two trees. We show that these ideas give us an almost linear time algorithm for approximating TED within a factor of at most $O(\sqrt{n})$.

Theorem 7.4 [restated informally]. *TED admits an $\widetilde{O}(n)$ time algorithm with approximation factor $O(\sqrt{n})$.*

4 FOREST EDIT DISTANCE

The forest edit distance problem extends the definition of edit distance to ordered forests. We use forest edit distance as an intermediary problem in our solution. Previous algorithms such as that of Zhang and Shasha [39] also use it as a subproblem for solving tree edit distance. In the forest edit distance problem (FED) the goal is to transform the first forest into the second forest, using the basic

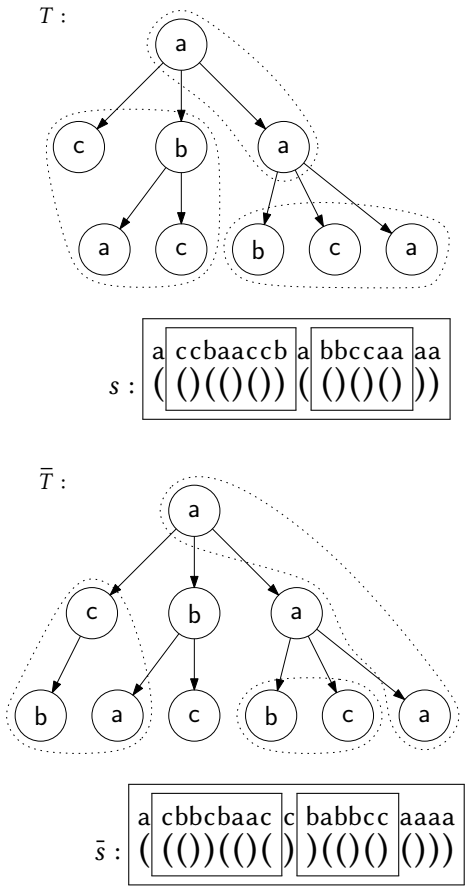


Figure 8: The synchronous decomposition of T and $\overline{T}$ is illustrated in this example. For T with 9 nodes and $\Delta = 3$, we partition T into 3 parts of size Δ. This decomposition is shown via rectangles for the representations.

operations of TED. Additionally, we assume that we are given all of the tree edit distances between any two pairs of nodes as input. That is, for any pair of nodes u and $\overline{u}$, $\text{ted}(u, \overline{u})$ is available. Recall that in $\text{ted}(u, \overline{u})$ we assume that u is mapped to $\overline{u}$.

Forest Edit Distance (FED)
Input: two ordered forests s and $\bar{s}$ of total size n and the ted's between any two pairs of nodes in s and $\bar{s}$.
Output: a sequence of operations that transforms s into $\bar{s}$ with the minimum length ($\text{fed}(s, \bar{s})$).

In Section 4.1, we briefly review how the technique of [33] solves FED in time $O(n + \mathsf{d}_{\max}{}^2)$ when the distance between the two ordered forests is bounded by $\mathsf{d}_{\max}$. Based on this idea, in Section 4.2, we provide a data structure FEDDS that for two given ordered forests s and $\bar{s}$, approximates the forest edit distance between any two proper subforests of them. We use FEDDS in Section 5 to approximate the forest edit distance between several subforests with a running time almost equal to the runtime of computing only one fed.

4.1 An $O(n + \mathsf{d}_{\max}{}^2)$ Time Solution

In this section, we review the solution of [33] for solving FED for two given ordered forests s and $\bar{s}$ of total size n with distance at most $\mathsf{d}_{\max}$. We use this solution in Section 4.2 to build our data structure for approximating the edit distances between subforests. In the following, first we present a classic $O(n^2)$ time algorithm for FED, and then we show how the time complexity improves to $O(n + \mathsf{d}_{\max}{}^2)$ in the algorithm of [33]. Let $\mathsf{m}(i)$ and $\overline{\mathsf{m}}(j)$ be the indices of the matched parentheses of $s[i]$ and $\bar{s}[j]$, respectively. Recall that ted's between all pairs of nodes are given. We compute $\text{fed}(s[1, i], \bar{s}[1, j])$'s via a dynamic program with the following update rule.

$$\text{fed}(s[1,\, i],\, \bar{s}[1,\, j]) = \min \begin{cases} \text{fed}(s[1,\, i-1],\, \bar{s}[1,\, j]) + \text{isClosing}(s[i]) & \text{if } i > 0, \\ \text{fed}(s[1,\, i],\, \bar{s}[1,\, j-1]) + \text{isClosing}(\bar{s}[j]) & \text{if } j > 0, \\ \text{fed}(s[1,\, \mathsf{m}(i)-1],\, \bar{s}[1,\, \overline{\mathsf{m}}(j)-1]) & \text{if isClosing}(s[i]) \\ +\text{ted}(s[\mathsf{m}(i),\, i],\, \bar{s}[\overline{\mathsf{m}}(j),\, j]) & \text{\& isClosing}(\bar{s}[j]). \end{cases}$$

Using this update rule, we compute $\text{fed}(s[1, i], \bar{s}[1, j])$'s for all $1 \le i \le |s|$ and $1 \le j \le |\bar{s}|$. Finally, $\text{fed}(s[1, i], \bar{s}[1, j])$ where $i = |s|$ and $j = |\bar{s}|$ is the forest edit distance between s and $\bar{s}$. The running time of this algorithm is $O(n^2)$.

It has been shown that ted reduces to almost $O(n^2)$ instances of fed [39]. Hence, using this algorithm, we can compute the ted between s and $\bar{s}$ in time $O(n^4)$. A more careful analysis improves the time complexity to $O(n^2 h\bar{h})$ [39], where h ($\bar{h}$) is the height of T ($\overline{T}$). Also, note that FED is a generalization of the edit distance (ED) problem since we can convert an input of ED into an input of FED. To do this, we replace any character with a pair of parentheses with the same label. For example, ed("tgcat", "atcct") = $\text{fed}(\overset{\text{ttggccaatt}}{\text{“()()()()()”}}, \overset{\text{aattcccctt}}{\text{“()()()()()”}}) = 3$.

Landau and Vishkin [29] show how to solve ED in time $O(n + \mathsf{d}_{\max}{}^2)$ when the size of the solution is bounded by $\mathsf{d}_{\max}$. For two strings s and $\bar{s}$, they use the observation that if $s[i + 1, i + t] = \bar{s}[j + 1, j + t]$, then $\text{ed}(s[1, i], \bar{s}[1, j]) = \text{ed}(s[1, i + t], \bar{s}[1, j + t])$. They use a suffix tree to compute the queries of the following type in constant time: given two indices i and j of s and $\bar{s}$, what is the largest t such that $s[i + 1, i + t] = \bar{s}[j + 1, j + t]$.

Shasha and Zhang use a more in-depth and more technically involved analysis of the same idea to present an algorithm for the forest edit distance problem in time $O(n + \mathsf{d}_{\max}{}^2)$ [33]. In Appendix A of the full version of the paper, we recapitulate some of the ideas of their algorithm to solve FED in time $O(n + \mathsf{d}_{\max}{}^2)$.

Theorem 4.1 (Proved in [33]). *For two ordered forests s and $\bar{s}$ of total size n, we can find their forest edit distance in time $O(n + \mathsf{d}_{\max}{}^2)$, where $\mathsf{d}_{\max}$ is an upper bound on the solution size, and the tree edit distances between nodes are given in the input.*

Similar to [29], Shasha and Zhang [33] fill a $(2\mathsf{d}_{\max} + 1) \times (\mathsf{d}_{\max} + 1)$ array in time $O(\mathsf{d}_{\max}{}^2)$ that has all the information of the $n \times n$ dynamic programming table. This array for any k and d stores the largest i such that $\text{fed}(s[1, i], \bar{s}[1, i+k]) \le d$. Using this array, we can find $\text{fed}(s[1, i], \bar{s}[1, j])$ for any i and j in time $O(\log \mathsf{d}_{\max})$ by doing

a binary search on the diagonal $k = j - i$ to find $\text{fed}(s[1,i], \bar{s}[1,j])$, if the distance is at most $\mathsf{d}_{\max}$ or report $\text{fed}(s[1,i], \bar{s}[1,j]) > \mathsf{d}_{\max}$ if the binary search could not find it. In Section 4.2, we use Theorem 4.1 to approximate the forest edit distances between all proper subforests of the two ordered forests.

4.2 Forest Edit Distance Data Structure

In this section, we design a data structure for two ordered forests s and $\bar{s}$ that approximates the forest edit distance between any two proper subforests of them. We call this data structure FEDDS. Recall that, for an ordered forest s, a subforest is the refined subsequence of the corresponding substring of s. Also, recall that a subforest is proper if no redundant closing parenthesis is present before the refinement. For a $1+\epsilon$ approximate solution ($\epsilon > 0$) the construction of FEDDS takes time $O((1/\epsilon^3)n^2)$, and then FEDDS can answer each query in time $\widetilde{O}(1)$.

We use k-bounded forest data structure, denoted by FEDDS_k, as an intermediate data structure. However, in Section 5, we directly use FEDDS_k. Using FEDDS_k constructed for two ordered forests s and $\bar{s}$, we can query an estimate of the fed between any two proper subforests of s and $\bar{s}$, if the distance is less than or equal to $k - 2\delta k$, or find out that the solution is more than k. For an arbitrary $\delta > 0$, FEDDS_k may have an additive error of up to $2\delta k$. The time complexity of constructing FEDDS_k is $O((1/\delta)^2 n^2)$, and it answers each query in time $O(\log k)$.

FEDDS reduces to FEDDS_k by losing a small error. For an arbitrary $\epsilon > 0$, we can construct FEDDS using a set of FEDDS_k's, where $k = (1+\epsilon)^i$ and $0 \le i \le \log_{1+\epsilon} n$. By using a suitable δ, we can adjust the approximation factor of FEDDS to be $1 + \epsilon$. The construction of FEDDS_k is as follows.

Let $1 \le k \le n$ be a given integer and δ be a given error threshold. To construct FEDDS_k for s and $\bar{s}$, we create a set of marked points for each of s and $\bar{s}$ consisting of indices divisible by $\lfloor \delta k \rfloor$. Therefore, we have $O(n/\delta k)$ marked points in total. Afterward, for every two marked points i and j in s and $\bar{s}$, respectively, we use the solution of Theorem 4.1 to compute the fed between $s[i, |s|]$ and $\bar{s}[j, |\bar{s}|]$. Recall that by using the solution of Theorem 4.1, we compute a two-dimensional $O(k) \times O(k)$ array which compactly stores all the fed's between forests $s[i, i']$'s and $\bar{s}[j, j']$'s, for any $i' > i$ and any $j' > j$, in cases where the distance is at most k. To answer a query to approximate $\text{fed}(s[i, i'], \bar{s}[j, j'])$, we decrease each of i and j to match one of the marked points. Then, we use the array we computed for shifted i and j, which are now marked indices, and output $\text{fed}(s[i, i'], \bar{s}[j, j'])$. The number of shifts of each of i and j to reach a marked point is at most δk, which results in a total additive error of at most $2\delta k$. In Lemma 4.2, we show that Algorithms 1 and 2 correctly construct and answer queries from FEDDS_k within the desired time complexity and additive error.

Lemma 4.2. *Let $\delta > 0$ be an arbitrarily small constant. For two ordered forests of total size n and an integer $k > 0$, Algorithm 1 constructs FEDDS_k in time $O((1/\delta^2)n^2)$. Henceforth, Algorithm 2 approximates the forest edit distance between any two proper subforests in time $O(\log k)$ within an additive error of $2\delta k$ if the answer is at most $k - 2\delta k$, or reports that the distance is more than k.*

As we discussed earlier, we use FEDDS_k's with $k = (1+\epsilon)^i$ for $0 \le i \le \log_{1+\epsilon} n$ to construct FEDDS. We set $\delta = \epsilon/2(1+\epsilon)$ to make the additive error be at most ϵ. We describe the construction and queries of FEDDS in Algorithms 3 and 4 and show their correctness in Theorem 4.3.

Algorithm 1: construct $\text{FEDDS}_k(s, \bar{s}, k, \delta)$

Data: two ordered forests s and $\bar{s}$, an upper bound k on the distance, and $\delta > 0$.
Result: $\text{FEDDS}_k(s, \bar{s})$.

```
for i = 1 to |s|, in steps of i ← i + ⌊δk⌋ do
    for j = 1 to |s̄|, in steps of j ← j + ⌊δk⌋ do
        compute fed(s[i, |s|], s̄[j, |s̄|]) for distances up to k,
        store the O(k) × O(k) array, and call it fed_{i,j}
```

Algorithm 2: query $\text{FEDDS}_k([i_l, i_r], [j_l, j_r])$

Data: two intervals associated with two proper subforests $t = s[i_l, i_r]$ and $\bar{t} = \bar{s}[j_l, j_r]$.
Result: $\text{fed}(t, \bar{t})$ with an additive error of at most $2\delta k$ if $\text{fed}(t, \bar{t}) \le k - 2\delta k$ or reports that $\text{fed}(t, \bar{t}) > k$.

1. find i, the index of the largest marked point in s less than or equal to i_l;
2. find j, the index of the largest marked point in $\bar{s}$ less than or equal to j_l;
3. find the answer in the array of $\text{fed}_{i,j}$ if exists and return it;
4. otherwise, report $\text{fed}(t, \bar{t}) > k$.

Algorithm 3: construct $\text{FEDDS}(s, \bar{s}, \epsilon)$

Data: two ordered forests s and $\bar{s}$, and $\epsilon > 0$.
Result: $\text{FEDDS}(s, \bar{s})$.

```
δ ← ε/2(1 + ε);
for k ∈ {1, 1 + ε, ..., (1 + ε)^{log_{1+ε} n}} do
    construct FEDDS_k(s, s̄, k, δ)
```

Algorithm 4: query $\text{FEDDS}([i_l, i_r], [j_l, j_r])$

Data: two intervals associated with two proper subforests $t = s[i_l, i_r]$ and $\bar{t} = \bar{s}[j_l, j_r]$.
Result: $\text{fed}(t, \bar{t})$ with an approximation factor of $1 + \epsilon$.

1. use the suffix tree to check whether $\text{fed}(t, \bar{t}) = 0$, and return 0 in this case;
2. **for** $k \in \{1, 1+\epsilon, \ldots, (1+\epsilon)^{\log_{1+\epsilon} n}\}$ **do**
3. query $\text{FEDDS}_k([i_l, i_r], [j_l, j_r])$ and return the answer if it does not report $\text{fed}(t, \bar{t}) > k$.

Theorem 4.3. *Let $\epsilon > 0$ be an arbitrarily small constant. For two ordered forests s and $\bar{s}$ of total size n, knowing the ted's between all pairs of parentheses of s and $\bar{s}$, Algorithm 3 constructs FEDDS in time $\widetilde{O}(n^2)$. Afterward, Algorithm 4 approximates the fed between any two subforests of s and $\bar{s}$ in time $\widetilde{O}(1)$ within an approximation factor of $1 + \epsilon$.*

5 A $1+\epsilon$ APPROXIMATION ALGORITHM FOR TED

One challenge that we face here is the depth of the computation and its effect on the approximation factor. More precisely, if the approximation factor of each level is $1+\epsilon'$, the overall approximation factor that accumulates in each level would be $1+\epsilon$, where $\epsilon = \Omega(h\epsilon')$. Here, h is the depth of the computation. Note that in a naïve approach, h can be as large as n.

In order to avoid this issue, we should have limited levels of computation in our algorithm. Besides this issue, computing ted for all node pairs one by one takes a total time of $\widetilde{O}(n^4)$ since we have $O(n^2)$ node pairs and computing the ted for each pair takes time $\widetilde{O}(n^2)$. There is a huge gap between $\widetilde{O}(n^4)$ and the desired $\widetilde{O}(n^2)$ time. In what follows, we show how applying the heavy-light decomposition of Sleator and Tarjan [34] enables us to overcome these two difficulties. This technique has been used in many previous works such as [6, 17, 19, 20, 27, 35] to design algorithms for trees of arbitrary height.

The heavy-light decomposition partitions the nodes of an ordered tree into a set of paths called *spines*. These spines may be long paths; however, the decomposition ensures that in a path from the root to any node (including leaves), we cross at most $O(\log n)$ spines. We compute the ted's between the nodes of two spines all at once instead of computing the ted's between every two nodes individually. Consequently, we keep the approximation factor small since the recursive depth of the computation is at most $O(\log n)$ independent of the heights of the input trees. We also improve the running time by constructing FEDDS once for the two spines and querying it several times to approximate the ted's between all pairs of nodes of two spines. This approach along with other techniques explained below helps us to keep the running time quadratic.

As mentioned, the main part of our algorithm is computing the ted's between the nodes of two spines, which we call *spine edit distance*. For two spines $S = (u_1, u_2, \dots)$ and $\bar{S} = (\bar{u}_1, \bar{u}_2, \dots)$, we assume the ted's between all pairs of nodes are given, except pairs of nodes between spines S and $\bar{S}$. In addition, using a bottom-up approach ensures that when we want to compute the $\text{ted}(u_i, \bar{u}_j)$, we already know the ted's between all of the deeper nodes in the spines.

In the following, we briefly describe how our algorithm computes $\text{ted}(u_i, \bar{u}_j)$. Let opt be an optimal solution of transforming the subtree of u_i to the subtree of $\bar{u}_j$. Note that by the definition of ted, u_i is transformed into $\bar{u}_j$ in opt. Suppose the next node in spine S after u_i which is not deleted in opt is $u_{i'}$, and it is mapped to a node w in the subtree of $\bar{u}_j$. Also, notice that w can be any node in the subtree of $\bar{u}_j$ and does not necessarily belong to spine $\bar{S}$. The cost of mapping u_i and its subtree into $\bar{u}_j$ and its subtree consists of six parts:

(1) mapping node u_i into node $\bar{u}_j$ (their opening and closing parentheses, excluding their inner parentheses),
(2) mapping the subforest of nodes before the opening parenthesis of $u_{i'}$ to the subforest of nodes before the opening parenthesis of w,
(3) mapping the subtree of node $u_{i'}$ to the subtree of node w,
(4) mapping the subforest of nodes after the closing parenthesis of $u_{i'}$ to the subforest of nodes after the closing parenthesis of w,
(5) deleting the path between u_i and $u_{i'}$, and finally
(6) inserting the path between $\bar{u}_j$ and w.

Recall that previously in Section 3, we formulate these six parts as $\text{ted}(u_i, \bar{u}_j) := \text{ted}(u_{i'}, w) + R + C + L$. Here, $\text{ted}(u_{i'}, w)$ is denoted by part (iii), R consists of parts (v) and (vi), C consists of parts (ii) and (iv), and L consists of part (i). Moreover, in the example of Figure 6, the cost of these six parts are 0, 1, 0, 1, 1, and 0, respectively.

In Section 5.2, we show how to compute the costs of all of these six parts in $\widetilde{O}(1)$ time for fixed $u_{i'}$ and w. Moreover, we show how to reduce the number of tuples $(u_i, \bar{u}_j, u_{i'}, w)$'s from $O(n^4)$ to $O(n^2)$ in order to reduce the time complexity to $\widetilde{O}(n^2)$.

Finally, in Section 5.3, we use the spine edit distance algorithm to design our algorithm to approximate the tree edit distance of the two input trees in time $\widetilde{O}(n^2)$ within an approximation factor of $1+\epsilon$.

5.1 Tree Decomposition

In this section, we define the heavy-light decomposition which we use in Sections 5.2 and 5.3. The content of this section can be skipped by the reader who is familiar with the heavy-light decomposition. However, some of our notations may be different from previous work. Let T be a tree, where u and v are two nodes of T, and v is a child of u. We call an edge $e = (u, v)$ *heavy*, if $\text{size}(v)$ has the maximum value among the children of u. In case of a tie, we choose v to be the right-most child of u with this property. We call all other edges *light*. We also define light and heavy nodes as follows. If a node v is a child of a node u and they are connected via a heavy edge, we call v heavy; otherwise, we call v light. Every node has exactly one heavy child, except leaves which have no children. For example, in Figure 9, edge (u_1, u_2) is heavy since $\text{size}(u_2) = 6$ is larger than $\text{size}(u_3) = 3$ and $\text{size}(u_4) = 1$. Moreover, edge (u_2, u_5) is light since $\text{size}(u_5) = 1$ is less than $\text{size}(u_6) = 4$. Similarly, node u_2 is heavy, and node u_3 is light. Now we define a spine as follows. For any light node u, it has a unique heavy child unless u is a leaf. We choose this heavy child and repeat the process until we end up in a leaf. This process defines a path in the tree from u to a leaf. We call this path (including u and the leaf) the *spine* of the light node u.

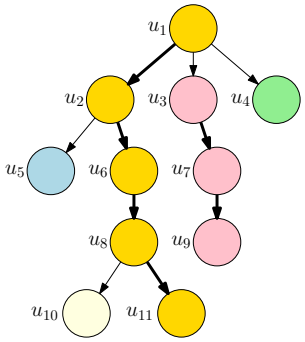


Figure 9: The heavy-light decomposition of a tree.

We iterate over all light nodes and find their spines. For example, in Figure 9, the spines are $(u_1, u_2, u_6, u_8, u_{11})$, (u_5), (u_{10}), (u_3, u_7, u_9), and (u_4). These spines partition the nodes of the tree into a number of paths. The most important property of this decomposition is that in any path from the root to any node, we pass through at most $O(\log n)$ spines.

Lemma 5.1 (Proved in [34]). *In a tree with a heavy-light decomposition, every path from the root to any other node pass through at most* $\lfloor \log_2 n \rfloor + 1$ *spines.*

In other words, along any path from the root to a leaf, there are at most $\lfloor \log_2 n \rfloor$ light edges since we change the spine only when we pass through a light edge. The number of nodes in the subtrees is divided by at least a factor of two each time we traverse a light edge. Therefore, we have at most $\lfloor \log_2 n \rfloor$ light edges in a path from the root to any node. In Section 5.2, we show how heavy-light decomposition helps us to solve the spine edit distance problem.

5.2 Spine Edit Distance

In this section, we solve spine edit distance for two spines $S = (u_1, u_2, \ldots, u_h)$ and $\bar{S} = (\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_{\bar{h}})$ in s and $\bar{s}$, respectively. We assume the tree edit distances between all node pairs of s and $\bar{s}$ are given, except node pairs of S and $\bar{S}$. Moreover, w.l.o.g. we assume u_1 and $\bar{u}_1$ are the roots of s and $\bar{s}$.

The first step in our algorithm is to use FEDDS to approximate the fed's between subforests of s and $\bar{s}$. However, constructing a FEDDS for s and $\bar{s}$ is not possible since not all ted's are known. To resolve this issue, we construct two FEDDS's instead of one as follows. Suppose the opening and closing parentheses of u_h, which is the last node of S are $s[i]$ and $s[i+1]$. Note that since u_h is a leaf node of the first tree, its opening and closing parentheses are consecutive in s. We construct one FEDDS between $s[1, i]$ and $\bar{s}$, and another FEDDS between $\text{rev}(s[i+1, |s|])$ and $\text{rev}(\bar{s})$. Here, rev reverses its input string. The reason for reversing strings is to ensure that our queries for part (iv) consist of proper subforests.

In the following, we assume that the goal is to approximate $\text{ted}(u_i, \bar{u}_j)$ for a node u_i of S and a node $\bar{u}_j$ of $\bar{S}$. Let opt be an optimal transformation with $\text{ted}(u_i, \bar{u}_j)$ operations. Moreover, since we compute $\text{ted}(u_i, \bar{u}_j)$'s in a bottom-up approach when we are computing $\text{ted}(u_i, \bar{u}_j)$, we already know $\text{ted}(u_{i'}, \bar{u}_{j'})$'s for $i < i' \le h$ and $j < j' \le \bar{h}$. Therefore, we know all the ted's between the nodes of the subtree of u_i and the nodes of the subtree of $\bar{u}_j$, expect the ted's between u_i and $\bar{u}_j$ itself.

Note that in $\text{ted}(u_i, \bar{u}_j)$, u_i is mapped to $\bar{u}_j$ by definition. To compute $\text{ted}(u_i, \bar{u}_j)$, we search for a $u_{i'}$, which is the first node after u_i in S which is not removed in opt. Node $u_{i'}$ is mapped to some node in the subtree of $\bar{u}_j$, namely w. Recall that for a fixed $u_{i'}$ and w, $\text{ted}(u_i, \bar{u}_j)$ is equal to the sum of these six parts:

(1) the cost of changing the label of u_i to the label of $\bar{u}_j$, if necessary,
(2) the cost of mapping the subforest of nodes before the opening parenthesis of $u_{i'}$ to the subforest of nodes before the opening parenthesis of w,
(3) the cost of mapping the subtree of node $u_{i'}$ to the subtree of node w,
(4) the cost of mapping the subforest of nodes after the closing parenthesis of $u_{i'}$ to the subforest of nodes after the closing parenthesis of w,
(5) the number of nodes between u_i and $u_{i'}$ which are deleted, and
(6) the number of nodes between $\bar{u}_j$ and w which are inserted.

The time complexity of an algorithm implementing this method without any additional ideas is $\widetilde{O}_\epsilon(n^4)$, and its approximation factor is $1 + \epsilon$. To improve the time complexity, we apply two ideas. Our first idea is based on Observation 5.1.

Observation 5.1. *Let π be a transformation with at most $\mathsf{d}_{\max}$ operations from s into $\bar{s}$. If π transforms $s[i]$ into $\bar{s}[j]$, then $|i - j| \le 2\mathsf{d}_{\max}$.*

Let d' be the sum of the costs of all parts except part (iii) in the optimal solution corresponding to $\text{ted}(u_i, \bar{u}_j)$. The first idea to improve the time complexity is that for a fixed u_i and $\bar{u}_j$, we have at most $O(d')$ possibilities for either $u_{i'}$ and w. To prove this upper bound for $u_{i'}$, note that we have a cost of $i' - i - 1$ in part (v) for removing all nodes in the path between u_i and $u_{i'}$ on spine S. Since we have at most d' operations in part (v), $i + 1 \le i' \le i + d' + 1$ holds. Therefore, $d' + 1$ is an upper bound on the number of possibilities of $u_{i'}$. In addition, we claim that for a fixed $u_{i'}$, we have at most $O(d')$ possibilities for w. Based on Observation 5.1, and since the cost of part (ii) is at most d', we conclude that the last closing parenthesis before the opening parentheses of w in $\bar{s}$ has at most $4d' + 1$ possibilities. Therefore, the opening parenthesis of w also has at most $O(d')$ possibilities. This idea reduces our running time from $\widetilde{O}(n^4)$ to $\widetilde{O}(n^2 d'^2)$ by reducing the number of possibilities of i' and w. The following idea reduces the number of possibilities of i and j to improve the running time to $\widetilde{O}(n^2)$.

Moreover, in our algorithm, we directly use FEDDS_k's instead of just using FEDDS. We want to use a FEDDS_k where k is near d' in the optimal transformation of the subtree of the subtree of u_i into the subtree of $\bar{u}_j$. Since we do not know the correct value of d', we try all values of $\{1, 1 + \epsilon, \ldots, (1 + \epsilon)^{\log_{1+\epsilon} n}\}$ as k. Then, for a value of k where $k/(1 + \epsilon) < d' \le k$, we use FEDDS_k to estimate the cost of parts (ii) and (iv). Recall that, in FEDDS_k we mark a number of points, and for each query of two subforests, we shift the starting indices to the left to match marked points. We use the observation that these shifts do not change the outcome of the query too much. Therefore, for two queries whose starting points in both s and $\bar{s}$ are near the same marked points, FEDDS_k outputs an identical value. To improve the running time of our algorithm, we use the same observation for computing ted's. More precisely, for a fixed k, when the opening and closing parentheses of u_i and $\bar{u}_j$ differ slightly such that immediate marked points on their right are the same, we claim that the corresponding ted values are relatively close. Due to the nested nature of opening and closing parentheses of nodes of spines, there are at most $O(n/\delta k) \cdot O(n/\delta k)$ relatively different ted values for a specific k. For this reason, we store $\text{ted}(u_i, \bar{u}_j)$'s in a lookup table. Before computing a $\text{ted}(u_i, \bar{u}_j)$, we check whether a close TED is already computed. If so, we get it from the lookup table; otherwise, we compute it.

Lemma 5.2 (spine edit distance). *Let $\epsilon > 0$ be an arbitrary constant, and s and $\bar{s}$ be two trees of total size n. Moreover, let S and*

$\bar{S}$ two spines of s and $\bar{s}$, respectively. If we have all the ted's between all pairs of nodes of s and $\bar{s}$ except between nodes of S and $\bar{S}$, we can compute ted's between nodes of S and $\bar{S}$ in time $\widetilde{O}(n^2)$, within an approximation factor of $1+\epsilon$.

5.3 Our Algorithm

In this section, we use the spine edit distance algorithm of Section 5.2 to find the desired approximation algorithm of TED between two trees. We can perform the algorithm of two spines for all pairs of spines in a bottom-up approach. This way, we already computed the ted's between nodes that we need as prerequisites in each step.

Note that $\text{ted}(r, \bar{r})$ where r and $\bar{r}$ are roots of s and $\bar{s}$, is not necessarily equal to $\text{ted}(s, \bar{s})$. The first one has an additional condition that the root of the first tree should be mapped to the root on the second tree. To compute $\text{ted}(s, \bar{s})$, we add a dummy root to each of two input trees, which is enclosing each of s and $\bar{s}$ with an additional pair of parentheses. Adding these dummy roots does not change the tree edit distance, and there is an optimal solution which maps the root of the first tree to the root of the second tree. Our algorithm is shown in Algorithm 5.

Algorithm 5: our-ted$(s, \bar{s}, \epsilon)$

Data: two trees s and $\bar{s}$, and $\epsilon > 0$.
Result: $\text{ted}(s, \bar{s})$ with an approximation factor of $1+\epsilon$.

```
add two dummy roots u_r and ū_r on top of s and s̄, respectively;
heavy-light decompose s and s̄ into spines;
for S in spines of s in a bottom-up order do
    for S̄ in spines of s̄ in a bottom-up order do
        run spine − edit − distance(s, s̄, S, S̄, ε) using the
        already stored distances and store new distances;
return ted(u_r, ū_r);
```

Theorem 5.3. *For two ordered trees s and $\bar{s}$ of total size n, we can compute their tree edit distance in time $\widetilde{O}(n^2)$ with an approximation factor of $1+\epsilon$.*

ACKNOWLEDGMENTS

The authors would like to thank Masoud Seddighin for helpful discussions and an anonymous reviewer for comments on an earlier version of the papers.

REFERENCES

[1] Amir Abboud. 2014. Hardness for Easy Problems. (2014). Presented at Satellite Workshop of ICALP (YR-ICALP).
[2] Tatsuya Akutsu, Daiji Fukagawa, and Atsuhiro Takasu. 2010. Approximating Tree Edit Distance Through String Edit Distance. *Algorithmica* 57, 2 (2010), 325–348.
[3] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. 1998. Marked Ancestor Problems. In *FOCS*. IEEE, 534–543.
[4] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. 2010. Polylogarithmic Approximation for Edit Distance and the Asymmetric Query Complexity. In *FOCS*. IEEE, 377–386.
[5] Alexandr Andoni and Krzysztof Onak. 2009. Approximating Edit Distance in Near-linear Time. In *STOC*. ACM, 199–204.
[6] Arturs Backurs, Piotr Indyk, and Ludwig Schmidt. 2017. Better Approximations for Tree Sparsity in Nearly-Linear Time. In *SODA*. SIAM, 2215–2229.
[7] Ziv Bar-Yossef, TS Jayram, Robert Krauthgamer, and Ravi Kumar. 2004. Approximating Edit Distance Efficiently. In *FOCS*. IEEE, 550–559.
[8] Tuğkan Batu, Funda Ergun, and Cenk Sahinalp. 2006. Oblivious String Embeddings and Edit Distance Approximations. In *SODA*. SIAM, 792–801.
[9] Philip Bille. 2005. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science* 337, 1 (2005), 217–239.
[10] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, and Saeed Seddighin. 2018. Approximating Edit Distance in Truly Subquadratic Time: Quantum and MapReduce. In *SODA*. SIAM, 1170–1189.
[11] Karl Bringmann, PawełGawrychowski, Shay Mozes, and Oren Weimann. 2018. Tree Edit Distance Cannot be Computed in Strongly Subcubic Time (unless APSP can). In *SODA*. SIAM, 1190–1206.
[12] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. 2016. Truly Sub-cubic Algorithms for Language Edit Distance and RNA-Folding via Fast Bounded-Difference Min-Plus Product. In *FOCS*. IEEE, 375–384.
[13] Peter Buneman, Martin Grohe, and Christoph Koch. 2003. Path Queries on Compressed XML. In *VLDB*. VLDB Endowment, 141–152.
[14] Horst Bunke and Kim Shearer. 1998. A Graph Distance Metric Based on the Maximal Common Subgraph. *Pattern Recognition Letters* 19, 3 (1998), 255–259.
[15] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucky, and Michael Saks. 2018. Approximating Edit Distance Within Constant Factor in Truly Sub-Quadratic Time. In *FOCS*. IEEE, 979–990.
[16] Sudarshan S. Chawathe. 1999. Comparing Hierarchical Data in External Memory. In *VLDB*. Morgan Kaufmann Publishers Inc., 90–101.
[17] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. 2007. An Optimal Decomposition Algorithm for Tree Edit Distance. In *ICALP*. Springer, 146–157.
[18] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. 2009. Compressing and Indexing Labeled Trees, with Applications. *J. ACM* 57, 1 (Nov. 2009), 4:1–4:33.
[19] Ofer Freedman, Paweł Gawrychowski, Patrick K. Nicholson, and Oren Weimann. 2017. Optimal Distance Labeling Schemes for Trees. In *PODC*. ACM, 185–194.
[20] Paweł Gawrychowski, Nadav Krasnopolsky, Shay Mozes, and Oren Weimann. 2017. Dispersion on Trees. In *ESA*. Dagstuhl, 40:1–40:13.
[21] Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press.
[22] Bernhard Haeupler, Aviad Rubinstein, and Amirbehshad Shahrasbi. 2019. Near-Linear Time Insertion-Deletion Codes and $(1+\epsilon)$-Approximating Edit Distance via Indexing. (2019). In *STOC*. ACM.
[23] Dov Harel and Robert E. Tarjan. 1984. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.* 13, 2 (1984), 338–355.
[24] Piotr Indyk. 2001. Algorithmic Applications of Low-distortion Geometric Embeddings. In *FOCS*. IEEE, 10–33.
[25] Rajesh Jayaram and Barna Saha. 2017. Approximating Language Edit Distance Beyond Fast Matrix Multiplication: Ultralinear Grammars Are Where Parsing Becomes Hard!. In *ICALP*. Dagstuhl, 19:1–19:15.
[26] Camille Jordan. 1869. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik* 70 (1869), 185–190.
[27] Philip N. Klein. 1998. Computing the Edit-Distance Between Unrooted Ordered Trees. In *ESA*. Springer, 91–102.
[28] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. 1998. Incremental String Comparison. *SIAM J. Comput.* 27, 2 (1998), 557–582.
[29] Gad M. Landau and Uzi Vishkin. 1986. Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm. In *STOC*. ACM, 220–230.
[30] Barna Saha. 2017. Fast & Space-Efficient Approximations of Language Edit Distance and RNA Folding: An Amnesic Dynamic Programming Approach. In *FOCS*. IEEE, 295–306.
[31] Stanley M. Selkow. 1977. The Tree-to-Tree Editing Problem. *Inform. Process. Lett.* 6, 6 (1977), 184–186.
[32] Bruce A. Shapiro and Kaizhong Zhang. 1990. Comparing Multiple RNA Secondary Structures Using Tree Comparisons. *Bioinformatics* 6, 4 (1990), 309–318.
[33] Dennis Shasha and Kaizhong Zhang. 1990. Fast Algorithms for the Unit Cost Editing Distance Between Trees. *Journal of Algorithms* 11, 4 (Dec. 1990), 581–621.
[34] Daniel D. Sleator and Robert E. Tarjan. 1983. A Data Structure for Dynamic Trees. *J. Comput. System Sci.* 26, 3 (June 1983), 362–391.
[35] Daniel D. Sleator and Robert E. Tarjan. 1985. Self-adjusting Binary Search Trees. *J. ACM* 32, 3 (1985), 652–686.
[36] Kuo Chung Tai. 1979. The Tree-to-Tree Correction Problem. *J. ACM* 26, 3 (July 1979), 422–433.
[37] Hélène Touzet. 2005. A Linear Tree Edit Distance Algorithm for Similar Ordered Trees. In *CPM*. Springer, 334–345.
[38] Michael S. Waterman. 1995. *Introduction to Computational Biology: Maps, Sequences and Genomes*. CRC Press.
[39] Kaizhong Zhang and Dennis E. Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262.