

$3 + \epsilon$ Approximation of Tree Edit Distance in Truly Subquadratic Time

Masoud Seddighin ✉

Institute for Research in Fundamental Sciences (IPM), School of Computer Science, Tehran, Iran

Saeed Seddighin ✉

Toyota Technological Institute at Chicago, IL, USA

Abstract

Tree edit distance is a well-known generalization of the edit distance problem to rooted trees. In this problem, the goal is to transform a rooted tree into another rooted tree via (i) node addition, (ii) node deletion, and (iii) node relabel. In this work, we give a truly subquadratic time algorithm that approximates tree edit distance within a factor $3 + \epsilon$.

Our result is obtained through a novel extension of a 3-step framework that approximates edit distance in truly subquadratic time. This framework has also been previously used to approximate longest common subsequence in subquadratic time.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases tree edit distance, approximation, subquadratic, edit distance

Digital Object Identifier 10.4230/LIPIcs.ITCS.2022.115

Acknowledgements We would like to thank Amir Abboud for very helpful discussions. The second author was supported by a Google Research Gift.

1 Introduction

Edit distance is one of the most fundamental problems in combinatorial pattern matching. 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. Tree edit distance is a natural generalization of edit distance to rooted trees. In this work, our focus is on approximation algorithms for tree edit distance and we present the first constant factor algorithm that runs in truly subquadratic time.

Tree edit distance was first introduced by Selkow [24] in the late 70's. Since then, tree edit distance has found its applications in various areas such as computational biology [4, 17, 25, 28], structured data analysis (e.g., XML) [10, 14, 16], image analysis [11], and compiler optimization [15]. 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 [17, 18].

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 \bar{T} using the smallest number of edit operations. We assume that both trees T and \bar{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 modify the label of an existing node. It is an easy exercise to see that tree edit distance is equivalent to edit distance in the special cases that both trees are stars or paths.



© Masoud Seddighin and Saeed Seddighin;

licensed under Creative Commons License CC-BY 4.0

13th Innovations in Theoretical Computer Science Conference (ITCS 2022).

Editor: Mark Braverman; Article No. 115; pp. 115:1–115:22

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The computational aspect of the problem is also widely studied. Tai [26] 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 series of works to an $O(n^4)$ algorithm [29], and an $O(n^3 \log n)$ algorithm [19]. Finally, Demaine, Mozes, Rossman, and Weimann provide an $O(n^3)$ time algorithm [15]. The seminal work of Bringmann, Gawrychowski, Mozes, and Weimann [9] proves that the cubic running time barrier for weighted tree edit distance cannot be beaten by algorithms that use so-called decomposition strategy, unless finding all pairs shortest paths (APSP) in a graph admits a truly subcubic time solution and weighted k -clique¹ admits an $O(n^{k-\epsilon})$ time solution. Recently, Mao [22] propose a reduction of tree edit distance to max-plus product of bounded-difference matrices which in turn implies a truly sub-cubic time algorithm for unweighted tree edit distance.

For bounded variants, an $O(nd_{\max}^3)$ algorithm is proposed by Touzet [27] that runs in subcubic time when the distance between the two trees is small (d_{\max} here denotes an upper bound on the solution size). Also, Akmal and Jin [1] present an algorithm that computes the tree edit distance of two trees in $O(nd_{\max}^2)$. On the approximation front, Akutsu *et al.* [2] provide an $O(n^{3/4})$ approximation algorithm that runs in quadratic time. Also, a quadratic time algorithm with approximation factor $O(n^{2/3})$ follows from the algorithm of Touzet [27] by solving the problem for instances whose distance is smaller than $n^{1/3}$ and reporting a solution with cost $2n$ for instances with a distance of at least $n^{1/3}$. Recently, Boroujeni *et al.* [6] give an $\tilde{O}_\epsilon(n^2)$ time algorithm that approximates tree edit distance within a factor $1 + \epsilon$. They also obtain an $O(\sqrt{n})$ approximation algorithm that runs in time $\tilde{O}(n)$.

In this work, we give the first constant approximation algorithm for tree edit distance that runs in truly subquadratic time. More precisely, the approximation factor of our algorithm is $3 + \epsilon$ for any arbitrarily small constant $\epsilon > 0$ and the runtime of our algorithm is $O(n^{1.99})$ (Indeed we did not try to optimize the exponent of the runtime in favour of simplicity and clarity). Our result is obtained through a novel extension of a 3-step framework [5, 12] that approximates edit distance in truly subquadratic time. This framework has also been previously used to approximate longest common subsequence in subquadratic time [23].

1.1 Preliminaries

Tree edit distance. Let Σ be a fixed finite alphabet and T and \bar{T} be two rooted trees. We call a tree T *labeled*, if each node is labeled with a symbol from Σ . We also call T an ordered tree if a left-to-right order among siblings in T is given. Two trees are identical if their roots have the same label, the number of the children of the roots are equal, and the subtrees of the children of the roots are also identical and in the same order.

In the tree edit distance problem, we are given two trees T and \bar{T} and we wish to transform T into \bar{T} with the minimum number of primitive operations (given below) applied to the labeled trees:

- **Relabel:** change the label of a node v in T .
- **Delete:** delete a node v in T with parent v' , and replace it by the children of v . The order of the children remains the same. If v is the root, each child becomes a new root making the graph a forest.
- **Insert:** insert a node v as a child of v' in T and make v the parent of a consecutive subsequence of the children of v' .

¹ 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.

■ **Table 1** Previous results on tree edit distance. In the bounded TED problem, we are guaranteed that the distance between the two trees is bounded by d_{\max} .

Problem	Reference	Approximation Factor	Running Time
Our Results			
TED	Theorem 5	$3 + \epsilon$	$\tilde{O}(n^{1.99})$
Previous Work			
bounded TED	[27] (2005)	exact	$O(nd_{\max}^3)$
bounded TED	[6] (2019)	$1 + \epsilon$	$\tilde{O}(nd_{\max})$
bounded TED	[1] (2021)	exact	$O(nd_{\max}^2 \log n)$
TED	[27] [*] (2005)	$O(n^{2/3})$	$O(n^2)$
TED	[2] (2010)	$O(n^{3/4})$	$O(n^2)$
TED	[2] (2010)	$O(h_{\max})$	$O(n^2)$
TED	[6] (2019)	$1 + \epsilon$	$\tilde{O}(n^2)$
TED	[6] (2019)	$O(\sqrt{n})$	$\tilde{O}(n)$
TED	[26] (1979)	exact	$O(n^6)$
TED	[29] (1989)	exact	$O(n^4)$
TED	[19] (1998)	exact	$O(n^3 \log n)$
TED	[15] (2007)	exact	$O(n^3)$
TED	[22] (2021)	exact	$O(n^{2.9546})$

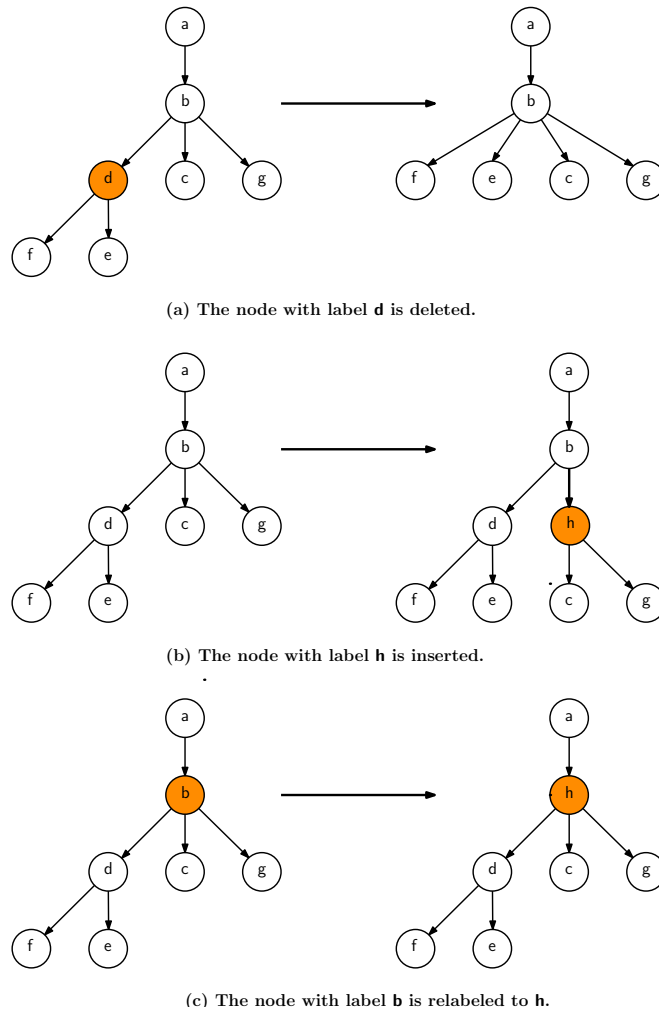
See Figure 1 for an example of these primitive operations. For two ordered and labeled trees T and \bar{T} , we denote their edit distance with $\text{ted}(T, \bar{T})$.

Representation. One common way to represent a tree is using nested parentheses. In this representation, each node v of tree T corresponds to a pair of opening and closing parentheses labeled with the same label as v , which encloses the children of v in the same order as they appear in T . Throughout the paper, we assume that s and \bar{s} correspond to the parentheses representation of T and \bar{T} respectively. Therefore, s and \bar{s} each consist of at most n pairs of labeled nested parentheses. See Figure 2 for an example.

We say a pair of opening and closing parentheses are twins if they correspond to the same node. For convenience, throughout the paper, we refer to the twin character of character i of string s by $\text{tw}(s_i)$. We use a similar notation for $\text{tw}(\bar{s}_i)$.

Note that for each of the operations defined in the previous paragraph, there is an equivalent string operation: insert is equivalent to inserting a pair of twin parentheses, delete is equivalent to deleting a pair of twin parentheses, and relabel is equivalent to changing labels of a pair of twin parentheses. Since most of our discussions in this paper are about string operations, it is useful to define some notations.

For a string r , we refer to its i^{th} element by r_i . Each symbol of s and \bar{s} is in the form of $\overset{\sigma}{(}$ or $\overset{\sigma}{)}$, where σ is a symbol of the alphabet. For any fixed transformation, we say a character in s or \bar{s} is matched, if it is either relabeled or remained unchanged in the transformation. Apart from the elements that are removed from s and the ones that are inserted into \bar{s} , all other characters are matched.



■ **Figure 1** Examples for delete, insert and relabel operations.

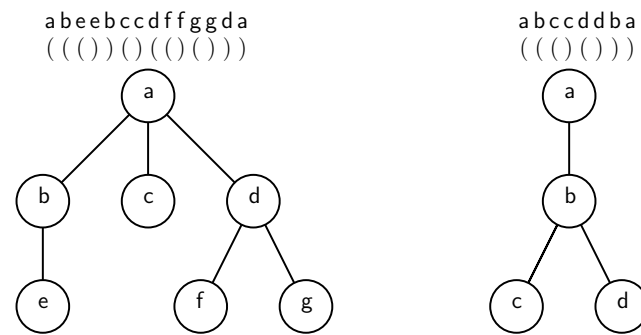
2 Our Techniques

Our algorithm is based on a framework that approximates edit distance in truly subquadratic time [5, 12]. This framework has also been used to approximate LCS in truly subquadratic time [23]. We begin by briefly explaining the previous ideas for approximating edit distance and then we show how we extend the framework to obtain a solution for TED.

2.1 Summary of Previous ED Techniques

When the edit distance of two strings s and \bar{s} is bounded by d_{\max} , the celebrated algorithm of Landau *et al.* [21] solves edit distance in time $O(|s| + |\bar{s}| + d_{\max}^2)$. Therefore, the problem is challenging only if the two strings are far ($\text{ed}(s, \bar{s}) = n^{1-o(1)}$)² otherwise the algorithm of Landau *et al.* [21] computes the solution in truly subquadratic time. The $3 + \epsilon$ factor approximation algorithm of Chakraborty *et al.* [12] for edit distance makes the following three steps to approximate edit distance of two strings of size n in truly subquadratic time.

² $\text{ed}(s, \bar{s})$ is the edit distance of s and \bar{s} .



■ **Figure 2** Each labeled rooted tree can be represented via a sequence of valid nested parentheses.

Step 1 (window-compatible solutions)

In the first step, they construct a set of *windows* W for string s , and a set of windows \overline{W} for string \bar{s} . Each window is basically a substring of an input string. Let k denote the total number of windows of $W \cup \overline{W}$. Although the windows may have different lengths, we assume for simplicity here that all the windows are of the same size d . When the solution size is $\Omega(n)$ (which is the case we wish to solve here), their construction maintains the property that $dk = O(n)$. The windows feature two key properties: 1) provided that the edit distances of the windows between W and \overline{W} are available, one can recover a $1 + \epsilon$ approximation of edit distance in time $\tilde{O}(n + k^2)$ via dynamic programming. 2) $k^2 \times d^2 \simeq O(n^2)$. That is, if we naively compute the edit distance of every pair of windows, the overall running time would still be asymptotically the same as that of the classic algorithm.

In order to obtain a solution for edit distance, it suffices to know the distances between the windows. However, Chakraborty *et al.* [12] show that knowing the distances between some of the window pairs is enough to obtain an approximately optimal solution for edit distance. Step 2 provides estimates for the distances of the windows which is approximately correct except for $O(k^{2-\Omega(1)})$ many pairs and Step 3 shows how this can be used to obtain a solution for edit distance.

Discretization simplifies the problem substantially. For a fixed $0 \leq \delta \leq 1$, they introduce a graph G_δ where the nodes correspond to the windows and an edge between window $w_i \in W$ and window $w_j \in \overline{W}$ means that $\text{ed}(w_i, w_j) \leq \delta n$. If we are able to construct G_δ for logarithmically different choices of δ , we can as well estimate the distances within a $1 + \epsilon$ factor for the windows. Therefore the problem boils down to constructing G_δ for a fixed given δ without computing the edit distance between all pairs of windows.

Step 2 (sparsification via triangle inequality)

Let v be a vertex of G_δ with a large enough degree. We discover all of the incident edges of v by computing its edit distance to the rest of the windows. Triangle inequality implies that every pair of windows in $N(v)$ has a distance bounded by $2\delta n$. Therefore by losing a factor 2 in the approximation, one can put all these edges in G_δ and not compute the edit distances explicitly. Although this does save some running time, in order to make sure the running time is truly subquadratic, we need to make a similar argument for paths of length 3 and thereby lose a factor 3 in the approximation. This method sparsifies the graph and what remains is to discover the edges of a sparse graph.

Step 3 (discovering the edges of the sparse graph)

Step 2 uses triangle inequality and discovers many edges between the vertices of G_δ . However, it may not discover all the edges completely. When in the remainder graph, the degrees are small (and hence the graph is sparse) triangle inequality does not offer an improvement and thus a different approach is required. Roughly speaking, Chakraborty *et al.* [12] subsample the windows of W into a smaller set S and discover all pairs of windows $w_i \in S$ and $w_j \in \overline{W}$ such that edge (i, j) is not discovered in Step 2. Next, they compute the edit distance of each pair of windows $(w_i, w_j), w_i \in W, w_j \in \overline{W}$ such that there exist two nearby windows (w_a, w_b) satisfying $w_a \in S, w_b \in \overline{W}$ and that the edge between w_a and w_b is missed in Step 2. The key observation is that even though this procedure does not discover all the edges, the approximated distances lead to an approximate solution for edit distance.

2.2 A Truly Subquadratic Time Algorithm for Tree Edit Distance

There are several variants of the above 3-step framework that have led to novel algorithms for many problems. For instance, prior to the work of Chakraborty *et al.* [12], Boroujeni *et al.* [5] utilize a quantum variant of Step 3 that in addition to the other two steps gives a quantum $3 + \epsilon$ approximation algorithm for edit distance. By modifying Step 2, Rubinstein *et al.* [23] give a truly subquadratic time algorithm that approximates LCS. Another variant of the framework given by Boroujeni *et al.* [7] also modifies Step 2 to obtain a $1 + \epsilon$ approximation algorithm for edit distance under mild assumptions. Several improvements are also made to the framework by adding extra steps that substantially improve the runtime for approximating edit distance [8, 20, 3]. We modify the first step of the framework to obtain a solution for tree edit distance.

Although tree edit distance is a strict generalization of edit distance, it maintains some key properties of edit distance which play an important role in the subquadratic time algorithm. Most importantly, tree edit distance satisfies triangle inequality. That is, given three trees T_1, T_2 , and T_3 , the tree edit distance of T_1 and T_2 plus the tree edit distance of T_2 and T_3 is an upper bound on the tree edit distance of T_1 and T_3 . Moreover, while the fastest known algorithm for tree edit distance requires $O(n^{2.9546})$ time [22], there is a $1 + \epsilon$ approximation algorithm for edit distance that runs in quadratic time if ϵ is constant [6]. The runtime of this algorithm improves to truly subquadratic if the solution size is truly sublinear [6].

While the above features are necessary for applying the framework of Chakraborty *et al.* [12], the most important part of their algorithm fails for tree edit distance. More precisely, in Step 1, Chakraborty *et al.* [12] use the concept of windows to divide the problem into smaller subproblems. This obviously fails for tree edit distance and as we discuss in the following, the concept of windows as defined by Chakraborty *et al.* [12] falls short of the purpose of Step 1.

The main issue with windows is that if one character falls within a window but its twin character is not included in the same window, then we have to remove the redundant character and we pay a price of 1 anyway. To see this, consider an example where for one of the trees the string representation is $((\dots())\dots)$ which is made by only having nested parentheses. In this example, the only possible window that covers the first character in a meaningful way is the one that covers the entire string. Therefore, if we use a window size which is substantially smaller than the string size we would not be able to find a window-compatible solution (a class of solutions that the algorithm of Chakraborty *et al.* [12] aims to find) that approximates the tree edit distance within a bounded factor. To address this issue, we introduce the notion of *super-windows* which are particularly designed to cover such parentheses.

A super-window is made of a left sub-window and a right sub-window. Each of the sub-windows covers a continuous interval of characters of an input string. For each super-window, we only take into account the characters whose twin characters are also covered by that super-window. However, it may be possible that the twin character of a character is included in a different sub-window of a super-window. We call the other characters *redundant* and ignore them in our computations. In our new Step 1, along with the windows, we also make a set of super-windows. We denote the set of windows by W and \bar{W} for s and \bar{s} and the super-windows by Q and \bar{Q} for s and \bar{s} .

Before we discuss the construction of the windows and super-windows, we would like to discuss how this new concept changes the algorithm. The most challenging part of the algorithm is Step 1 in which we are required to construct the windows and super-windows in a way that a window-compatible solution approximates the tree edit distance. Our construction, and in general any construction that allows for a desirable window-compatible solution, has to be *adaptive* and make the windows and super-windows based on the trees. This is unlike edit distance where the starting point and ending points of the windows are the same for any string of length n and does not depend on the strings. We also revise Step 3 of the algorithm to also mitigate the error for the distance of the super-windows which we have not approximated well-enough in Step 2. Another consequence of this new concept is that we need a new dynamic program to find the optimal window-compatible solution for the two trees that requires a slightly larger runtime. What remains intact in our algorithm is Step 2 in which we use triangle inequality to approximate the tree edit distances for many pairs.

Let us bring an example on why non-adaptive constructions fall short of our purpose. Fix a constant $c > 1$ and suppose that we know all the windows and sub-windows are of length smaller than w_{\max} . Now consider the following collection of instances: for every $1 \leq i \leq n/6w_{\max}$, we define instance \mathbb{I}_i as follows: both s and \bar{s} are of form $B_1B_2B_3B_4$, where $|B_1| = n/3$, $|B_2| = 2iw_{\max}$, $|B_3| = n/3$, and $|B_4| = n/3 - |B_2|$. Furthermore, we know that B_1 only consists of opening parentheses whose twin pairs are in B_3 . Also, symbols in B_2 and B_4 constitute valid sequences of nested parentheses. String \bar{s} is similar to s , except that it has different symbols for $n/6c$ of the parentheses in blocks B_1 and B_3 . Therefore, we have $\text{ted}(s, \bar{s}) = n/6c$. On the other hand, any window-compatible solution with approximation factor c must contain super-windows that cover at least $n/3 - cn/6c = n/6$ of the twin pairs in B_1 and B_3 . Therefore at least $n/6w_{\max}$ super-windows are needed to produce a window-compatible solution with the desired approximation factor for \mathbb{I}_i . On the other hand, note that the size of B_2 in \mathbb{I}_i and \mathbb{I}_j differ by at least $2w_{\max}$. Therefore, none of the super-windows that are used to cover B_1 and B_3 in \mathbb{I}_i are useful for instance \mathbb{I}_j . Therefore, since a non-adaptive construction must provide a desirable window-compatible solution to all $n/6w_{\max}$ instances, it must contain at least $(n/6w_{\max})(n/6w_{\max}) = n^2/36w_{\max}^2$ super-windows. Such a construction is obviously not desirable since even if we compute the tree edit distance once for each window/super-window, the total runtime exceeds quadratic. A similar example shows that in order to obtain a constant factor approximation, we need to consider windows of different lengths and even super-windows with varying length for its sub-windows. This is again in contrast to the case of edit distance since for edit distance, by losing a factor of $3 + \epsilon$ in the first step, we can use windows of equal length.

In the following, we give a brief explanation of how we construct the windows. For brevity, we only explain how we construct the windows and the super-windows of s . We then construct the windows and super-windows of \bar{s} in a similar way (but we use different parameters for \bar{s}). Our construction depends on parameters w_{\min} , w_{\max} , ϵ_{win} and γ . Define

$$L = \langle w_{\min}, w_{\min}(1 + \epsilon_{\text{win}}), w_{\min}(1 + \epsilon_{\text{win}})^2, \dots, w_{\max} \rangle, \mathcal{G} = \langle 1, 1 + \gamma, 1 + 2\gamma, \dots, n \rangle,$$

and denote the i^{th} element of L and \mathcal{G} respectively by L_i and \mathcal{G}_i . This implies that

$$|L| = O(\log_{1+\epsilon_{\text{win}}} w_{\text{max}}/w_{\text{min}}) \quad \text{and} \quad |\mathcal{G}| = O(n/\gamma).$$

Now, we add the following windows and super-windows to W and Q :

- For every $1 \leq i < |L|$ and every $1 \leq j < |\mathcal{G}|$, we verify if there exists a valid window $w = s[a, b]$ such that $b - a + 1 \in [L_i, L_{i+1})$ and $a \in [\mathcal{G}_j, \mathcal{G}_{j+1})$. If so, we add one such window to W . In the case that there are multiple such windows, we choose the window with the rightmost starting index. If there are multiple such windows starting at that index, we choose the shortest one.
- For every $1 \leq i, j < |L|$ and every $1 \leq k < |\mathcal{G}|$, we verify if there exists a valid super-window $q = s[(a, b)(b', a')]$ such that $b - a + 1 \in [L_i, L_{i+1})$, $a' - b' + 1 \in [L_j, L_{j+1})$, and $a \in [\mathcal{G}_k, \mathcal{G}_{k+1})$. If so, we add one such super-window to Q . In case there are multiple options, we choose the super-window with the largest starting index. If there are multiple such super-windows, we choose the one with the shortest left sub-window.

Also, we verify if there exists a valid super-window $q = s[(a, b)(b', a')]$ such that $b - a + 1 \in [L_i, L_{i+1})$, $a' - b' + 1 \in [L_j, L_{j+1})$ and $a' \in [\mathcal{G}_k, \mathcal{G}_{k+1})$. If so, we add one of them to Q . In case there are multiple such super-windows, we choose the super-window with the smallest ending index. If there are multiple such super-window, we add the one with the shortest right sub-windows.

It follows from our construction that $|W| = O(|L| \cdot |\mathcal{G}|) = O((\log_{1+\epsilon_{\text{win}}} w_{\text{max}}/w_{\text{min}})n/\gamma)$ and $|Q| = O(|L|^2 \cdot |\mathcal{G}|) = O((\log_{1+\epsilon_{\text{win}}} w_{\text{max}}/w_{\text{min}})^2 n/\gamma)$. We use a similar process to construct the set of windows (\bar{W}) and the set of super-windows (\bar{Q}) for \bar{s} . The only difference is that we use a different parameter $\bar{w}_{\text{max}} > w_{\text{max}}$ and similarly define

$$\bar{L} := \{w_{\text{min}}, w_{\text{min}}(1 + \epsilon_{\text{win}}), w_{\text{min}}(1 + \epsilon_{\text{win}})^2, \dots, \bar{w}_{\text{max}}\}$$

instead of L in the construction.

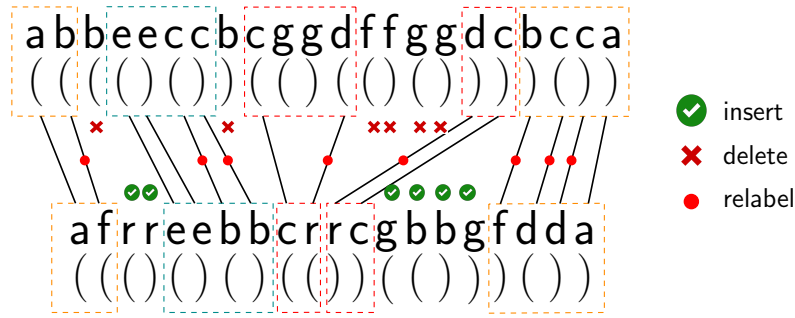
Inspired by the frameworks of [5, 12] we define a window-compatible solution in the following way:

► **Definition 1.** Let $S = \langle w_1, w_2, \dots, w_x \rangle$ and $\bar{S} = \langle \bar{w}_1, \bar{w}_2, \dots, \bar{w}_x \rangle$ be two sequences of size x of non-overlapping windows/super-windows from $W \cup Q$ and $\bar{W} \cup \bar{Q}$. We call a transformation of s into \bar{s} window-compatible with respect to S and \bar{S} , if

- All matched characters of s are in the windows and super-windows of S . Similarly, all matched characters of \bar{s} are in the windows and super-windows of \bar{S}
- Every matched character of \bar{s} which is in some window/super-window \bar{w}_i belongs to window/super-window w_i of s prior to the transformation.

We call a transformation window-compatible, if it is window-compatible with respect to at least one pair of sequences of non-overlapping windows/super-windows of $W \cup Q$ and $\bar{W} \cup \bar{Q}$.

Notice that Definition 1 is defined the same way as in [5], except that it incorporates the super-windows in the solution as well. See Figure 3 for an example of a window-compatible transformation. The main technical part of Step 1 of our algorithm is to prove that our construction leads to an almost optimal window-compatible transformation. More precisely, we prove in the full version that by setting the right values for parameters w_{min} , w_{max} , \bar{w}_{max} , ϵ_{win} and γ , one can bound the error of window-compatible solutions by any arbitrarily small multiplicative factor.



■ **Figure 3** An example of a window-compatible transformation.

► **Lemma 2.** *Let T and \bar{T} be two ordered and labeled trees and let s and \bar{s} be their representations with nested parentheses. In addition, let*

$$\varepsilon_1 = (8\varepsilon_{win}) + (190\gamma/w_{max}) + (54w_{min}/w_{max}) + (6w_{max}/\bar{w}_{max}).$$

Then, there exists a window-compatible transformation from s to \bar{s} whose cost is bounded by $\text{ted}(T, \bar{T}) + n\varepsilon_1$.

Before we discuss the proof of Lemma 2, we would like to bring two observations here. First, similar to edit distance, the challenge of approximating tree edit distance is when the solution size is large ($n^{1-o(1)}$). In this case, one can set the variables in a way that the total number of windows and super-windows ($O(k)$) times the maximum window/super-window size is bounded by $n^{1+o(1)}$. Moreover, if the tree edit distance between every pair of windows/super-windows is available, one can determine the optimal window-compatible solution in time $O(n + k^4)$ via a dynamic program. In this dynamic program, we maintain two DP tables D and T . $D[i][j]$ corresponds to a window/super-window w_i of $W \cup Q$ and a window/super-window w_j of $\bar{W} \cup \bar{Q}$ and stores the smallest cost we need to pay in order to transform $s_{1,x}$ into $\bar{s}_{1,y}$ where x and y are the ending positions of w_i and w_j , respectively. T is a four dimensional table and $T[i][i'][j][j']$ corresponds to two super-windows $q_i \in Q$ and $q_j \in \bar{Q}$ and two windows/super-windows $w_{i'} \in W \cup Q$ and $w_{j'} \in \bar{W} \cup \bar{Q}$ and stores the smallest cost we need to pay in order to transform $s_{x,x'}$ into $\bar{s}_{y,y'}$ where x is the index of the first character after the left-subwindow of q_i , x' is the last character of $w_{i'}$, y is the index of the first character after the left sub-window of q_j and y' is the last character of $w_{j'}$. In this case $w_{i'}$ should be nested under q_i and $w_{j'}$ should be nested under q_j . We show in the full version how we can recursively compute the values of the tables D and T and that this finds the optimal window-compatible solution in time $O(n + k^4)$ where k is the total number of windows and super-windows in W, \bar{W}, Q, \bar{Q} .

► **Lemma 3** (proven in the full version). *Let M be a $k \times k$ matrix that contains the pair-wise distances of the windows and super-windows. Based on M , one can find the optimal window-compatible transformation from s into \bar{s} in time $O(n + k^4)$.*

Thus, by keeping the number of windows and super-windows bounded by $n^{1/4-\Omega(1)}$ and approximating the pair-wise tree edit distances between the windows/super-windows, we can approximate the tree edit distance of the input trees in truly subquadratic time. All that is required is to spend time less than $O(w_{max}^2)$ for each pair of windows/super-windows on average. This is handled in Steps 2 and 3 by using triangle inequality and error detection.

The proof of Lemma 2 is based on an algorithmic thought experiment. We fix an optimal solution and aim to come up with windows/super-widows that give us a window-compatible solution with almost the same transformation cost. In the first step, we cover the matched

characters of s by a set of windows and super-windows with a greedy algorithm. These windows and super-windows may not belong to sets W and Q since they may be too short. In the second step, we show that by ignoring the short windows/super-windows we only miss a small number of matched characters of s . This is due to the fact that the total number of windows/super-windows we make is bounded and thus ignoring the small windows/super-windows does not incur a large cost. In the last step, we make equivalent windows/super-windows for \bar{s} and show that one can modify the windows/super-windows in a way that they fall within W, Q and \bar{W}, \bar{Q} and the cost of the transformation remains approximately intact. Our modifications may turn super-windows into windows that comes with an extra cost to ignore the redundant characters. A detailed explanation of this proof is given in the full version. Lemma 2 along with Lemma 3 concludes Step 1 of our algorithm. In Steps 2 and 3, we approximate the pairwise distances of the windows/super-windows and finally, we run the dynamic program to find an approximately optimal window-compatible solution.

We borrow Step 2 of our algorithm from previous work, stated as Lemma 4. Since Lemma 4 only depends on triangle inequality (which holds for edit distance as well as tree edit distance) we do not make any major modifications to the algorithm. The only difference is that instead of using the exact $O(n^2)$ time algorithm for edit distance, we use the $\tilde{O}_\epsilon(n^2)$ algorithm of Boroujeni *et al.* [6] that approximates tree edit distance within a factor $1 + \epsilon$. For the sake of completeness, we bring the full version.

► **Lemma 4** (Follows implicitly from [5] and [12]). *For any $1 \leq \Delta \leq k$ and any constant $\epsilon_M > 0$, there exists an algorithm that runs in time $\tilde{O}_{\epsilon_M}(k^2 \bar{w}_{\max}^2 / \Delta + k^3 / \Delta)$ and outputs a matrix $M : (|W| + |Q|) \times (|\bar{W}| + |\bar{Q}|) \rightarrow \mathbb{N} \cup \{0\}$ such that with probability at least $1 - 1/n^2$ we have:*

- *For each window/super-window $w_i \in W \cup Q$ and window/super-window $w_j \in \bar{W} \cup \bar{Q}$, $M[i][j]$ estimates the tree edit distance of w_i and w_j .*
- *M never underestimates the distances.*
- *For each window/super-window $w_i \in W \cup Q$ the number of windows/super-windows $w_j \in \bar{W} \cup \bar{Q}$ such that M does not estimate their distance within a factor $3 + \epsilon_M$ is bounded by Δ . Also, For each window/super-window $w_j \in \bar{W} \cup \bar{Q}$ the number of windows/super-windows $w_i \in W \cup Q$ such that M does not estimate their distance within a factor $3 + \epsilon_M$ is bounded by Δ .*

In the last step, we refine M and transform it into a matrix M' and show that if we use the values of M' for finding the optimal window-compatible solution, we would be able to approximate the tree edit distance within an approximation factor of $3 + \epsilon$. Recall that by Lemma 4, we know that in matrix M , for each window or super-window w of $W \cup Q$, the number of windows/super-windows whose distances to w is not estimated within a factor $3 + \epsilon_M$ is upper-bounded by Δ . Let us call these anomalies *overestimated distances*. In the third step, our goal is to correct a subset of overestimated distances which leads to a $3 + \epsilon$ approximate solution.

Our approach is similar to the method used by Chakraborty *et al.* [13] except that we also mitigate the error caused by the overestimated distances of super-windows. Roughly speaking, this method is consisted of two phases. In the first phase which we call the *random sampling* phase, we construct a subset R of the windows/super-windows of $W \cup Q$ by adding each window/super-window to R with some probability $0 < p < 1$, where the value of p is later tuned to optimize the running time and the approximation factor. For the rows in R , we refine their distances to all the windows/super-windows of $\bar{W} \cup \bar{Q}$ by approximating the tree edit distances via the algorithm of [6]. Thereafter, we use R as a basis to refine the rest of the overestimated distances in the next phase.

We call the second phase *nearby searching*. The method of Chakraborty *et al.* [13] runs as follows: Let (w, \bar{w}) be a pair of windows such that w belongs to R , and \bar{w} is one of the at most Δ windows whose distance to w has been overestimated in Step 2. Consider circles C and \bar{C} of radius ℓ centered at the starting indices of w and \bar{w} , respectively. For every pair of windows (w_x, w_y) such that the w_x falls within C and w_y falls within \bar{C} , we update their distance by computing it from scratch using the method of Boroujeni *et al.* [6]. Chakraborty *et al.* [13] show that when Δ is truly sublinear in k , by the right choice of p and ℓ , the algorithm runs in truly subquadratic time and obtains a $3 + \epsilon$ approximate solution for edit distance. Our method follows a similar roadmap except that we also incorporate the concept of super-windows in both the random sampling and nearby searching phases. Thus, for a super window, instead of drawing a single circle, we draw two circles and consider any super-window whose sub-windows fall within the drawn circles. This brings more details to the analysis which we discuss in the full version.

► **Theorem 5.** *For any $0 < \epsilon < 1$, there exists an algorithm with runtime $\tilde{O}_\epsilon(n^{1.99})$ that approximates the tree edit distance of two trees of size at most n within a factor $3 + \epsilon$.*

Theorem 5 follows from the 3-step framework we discussed above. In the rest of this paper, we describe the first step of our algorithm. We refer the reader to the full version of the paper for details on Step 2 and Step 3 of the algorithm.

3 Step 1: Window-compatible Solutions

In this section, we give a detailed explanation of the first step of our algorithm. We begin by defining the concept of windows and super-windows in Section 3.1. We then explain how we construct the windows and super-windows in Section 3.2. Finally, in Section 3.3, we prove that due to our construction, window-compatible solutions are almost optimal for tree edit distance.

3.1 Overview: Windows and Super-windows

Window is a commonly used concept in recent advances in string similarity problems, especially edit distance. Typically, for a string s , a window is defined to be a substring of s with length significantly smaller than $|s|$. The idea of using windows for approximating edit distance is indeed based on a divide and conquer technique: construct a set of windows from each string and carefully approximate the edit distance of each pair of windows. Next, we utilize this information to obtain a good approximation for edit distance. This reduces the task of approximating the edit distance between two long strings to approximating edit distance between many smaller windows.

We too make use of the idea of windows to approximate tree edit distance. However, this step is relatively more challenging in our algorithm as there are several limitations to be considered. For example, if an opening or closing parenthesis lies inside a window but its twin character is left outside the window, this incurs a cost to the approximation guarantee. Generally, twin parentheses may be too far from each other which may require us to construct very large windows. This signals that the subproblems that are formed in Step 1 of the framework should go beyond windows and thus a more complex structure is required. We address this issue by introducing the notion of super-windows which is used alongside windows in our algorithm.

Let r be a string. We define window $r[a, b]$ to be a substring of r which starts at position a and ends at position b (and therefore, the window is of length $b - a + 1$). In addition, define $r[(a, b), (b', a')]$ to be a super-window of r which is consisted of two sub-windows: The

Now, we add the following windows and super-windows to W and Q :

- For every $1 \leq i < |L|$ and every $1 \leq j < |\mathcal{G}|$, we verify if there exists a valid window $w = s[a, b]$ such that $b - a + 1 \in [L_i, L_{i+1})$ and $a \in [\mathcal{G}_j, \mathcal{G}_{j+1})$. If so, we add one such window to W . In the case that there are multiple such windows, we choose the window with the rightmost starting index. If there are multiple such windows starting at that index, we choose the shortest one.
- For every $1 \leq i, j < |L|$ and every $1 \leq x < |\mathcal{G}|$, we verify if there exists a valid super-window $q = s[(a, b)(b', a')]$ such that $b - a + 1 \in [L_i, L_{i+1})$, $a' - b' + 1 \in [L_j, L_{j+1})$, and $a \in [\mathcal{G}_x, \mathcal{G}_{x+1})$. If so, we add one such super-window to Q . In case there are multiple options, we choose the super-window with the largest starting index. If there are multiple such super windows, we choose the one with the shortest left sub-window.

Also, we verify if there exists valid a super-window $q = s[(a, b)(b', a')]$ such that $b - a + 1 \in [L_i, L_{i+1})$, $a' - b' + 1 \in [L_i, L_{i+1})$ and $a' \in [\mathcal{G}_x, \mathcal{G}_{x+1})$. If so, we add one of them to Q . In case there are multiple such super-windows, we choose the super-window with the smallest ending index. If there are multiple such super-window, we add the one with the shortest right sub-windows.

It follows from our construction that

$$|W| = O(|L| \cdot |\mathcal{G}|) = O((\log_{1+\epsilon_{\text{win}}} w_{\text{max}}/w_{\text{min}})n/\gamma)$$

and

$$|Q| = O(|L|^2 \cdot |\mathcal{G}|) = O((\log_{1+\epsilon_{\text{win}}} w_{\text{max}}/w_{\text{min}})^2 n/\gamma).$$

We use a similar process to construct the set of windows (\bar{W}) and the set of super-windows (\bar{Q}) for \bar{s} . The only difference is that we use a different parameter $\bar{w}_{\text{max}} > w_{\text{max}}$ and similarly define

$$\bar{L} := \{w_{\text{min}}, w_{\text{min}}(1 + \epsilon_{\text{win}}), w_{\text{min}}(1 + \epsilon_{\text{win}})^2, \dots, \bar{w}_{\text{max}}\}$$

instead of L in the construction. As mentioned, the values of $w_{\text{min}}, w_{\text{max}}$ and \bar{w}_{max} will be adjusted later. Throughout the paper, we let

$$k = \theta\left(\frac{n}{\gamma} (\log_{1+\epsilon_{\text{win}}} \bar{w}_{\text{max}})^2\right) \tag{1}$$

and therefore $|W| + |Q| + |\bar{W}| + |\bar{Q}| = O(k)$.

Before we proceed to Section 3.3, we recall the definition of window-compatible solutions.

► **Definition 1.** Let $S = \langle w_1, w_2, \dots, w_x \rangle$ and $\bar{S} = \langle \bar{w}_1, \bar{w}_2, \dots, \bar{w}_x \rangle$ be two sequences of size x of non-overlapping windows/super-windows from $W \cup Q$ and $\bar{W} \cup \bar{Q}$. We call a transformation of s into \bar{s} window-compatible with respect to S and \bar{S} , if

- All matched characters of s are in the windows and super-windows of S . Similarly, all matched characters of \bar{s} are in the windows and super-windows of \bar{S}
- Every matched character of \bar{s} which is in some window/super-window \bar{w}_i belongs to window/super-window w_i of s prior to the transformation.

We call a transformation window-compatible, if it is window-compatible with respect to at least one pair of sequences of non-overlapping windows/super-windows of $W \cup Q$ and $\bar{W} \cup \bar{Q}$.

3.3 Window-compatible Solutions are Approximately Optimal

In this section, we prove Lemma 2. At a high level, Lemma 2 implies that there exists a window-compatible solution whose cost is close to tree edit distance of T and \bar{T} .

► **Lemma 2.** *Let T and \bar{T} be two ordered and labeled trees and let s and \bar{s} be their representations with nested parentheses. In addition, let*

$$\varepsilon_1 = (8\varepsilon_{win}) + (190\gamma/w_{max}) + (54w_{min}/w_{max}) + (6w_{max}/\bar{w}_{max}).$$

Then, there exists a window-compatible transformation from s to \bar{s} whose cost is bounded by $\text{ted}(T, \bar{T}) + n\varepsilon_1$.

In the rest of this section, we prove Lemma 2. Let opt be a solution that transforms s into \bar{s} with the minimum number of operations and $|\text{opt}|$ denotes its cost. Our general approach to prove Lemma 2 is as follows: We fix an optimal transformation of s into \bar{s} . In the first step, we cover the matched pairs of s by a set of windows and super-windows. These windows and super-windows may not belong to sets W and Q since they may be too short. In the second phase, we show that by ignoring the short windows/super-windows we only miss a small number of matched pairs of s . In the last step, we make equivalent windows/super-windows for \bar{s} and show that one can modify the windows/super-windows in a way that they fall within W, Q and \bar{W}, \bar{Q} and the cost of the transformation remains approximately intact.

Fix an optimal solution for the tree edit distance of the two trees and let $(\alpha_1, \alpha'_1), (\alpha_2, \alpha'_2), \dots$ be pairs of indices in s that are not removed in the transformation. We assume that these pairs are sorted according to their first element (note that some of these pairs might be nested and thus the sortedness does not carry over to the second elements). In addition, let $(\beta_1, \beta'_1), (\beta_2, \beta'_2), \dots$ be the indices of these twin pairs, after the fixed optimal transformation. It follows that these twin pairs have the same structure in both strings; that is, the relative positions of these parentheses are the same in both strings. For example, if (α_x, α'_x) is nested under (α_l, α'_l) , then the same holds for (β_x, β'_x) and (β_l, β'_l) .

We start by covering s with windows and super-windows of arbitrary lengths and arbitrary starting points. Our only goal for now is that the length of each window and each sub-window of a super-window does not exceed w_{max} . For this, we run Algorithm 1 on the matched pairs of s . Algorithm 1 constructs the windows as follows: starting from the first pair (α_1, α'_1) , we iterate over every matched pair of s one by one and verify if it is covered by a currently made window or super-window. If so, we move on to the next matched pair. Otherwise, let $(\alpha_\ell, \alpha'_\ell)$ be the first pair which is not covered by any window/super-window. We cover $(\alpha_\ell, \alpha'_\ell)$ in the following way:

- If $|\alpha'_\ell - \alpha_\ell| < w_{max}$, let p be the largest index such that (i) $p \geq \ell$, (ii) for $mx := \max\{\alpha'_\ell, \alpha'_{\ell+1}, \dots, \alpha'_p\}$ we have $mx - \alpha_\ell < w_{max}$ (keep in mind that the position of the closing parentheses in the list are not necessarily increasing), and (iii) $s[\alpha_\ell, mx]$ does not overlap with any currently added super-window. We add window $s[\alpha_\ell, mx]$ to the set of windows. Note that, it might be the case that $p = \ell$.
- If $|\alpha'_\ell - \alpha_\ell| \geq w_{max}$, we add super-window $s[(\alpha_\ell, \alpha_p)(\alpha'_p, \alpha'_\ell)]$ to the set of super-windows, where p is the maximum index such that the length of each sub-window of $s[(\alpha_\ell, \alpha_p)(\alpha'_p, \alpha'_\ell)]$ is at most w_{max} . Note that, (α_p, α'_p) is itself a matched pair. Also, it might be the case that $p = \ell$. $s[(\alpha_\ell, \alpha_p)(\alpha'_p, \alpha'_\ell)]$ does not overlap with any of the previous windows and super-windows due to our algorithm.

Let W_1 and Q_1 be the set of windows and super-windows made by Algorithm 1 for s . All the matched indices of s are covered by some window or super-window. The length of each window of W_1 and each sub-window of the super-windows of Q_1 is bounded by w_{max} .

■ **Algorithm 1** Covering Matched Pairs.

```

1 Input:  $(\alpha_1, \alpha'_1), (\alpha_2, \alpha'_2), \dots$ : pairs of matched twins.
2 Output: A set of non-overlapping windows/super-windows covering matched pairs.
3 Function Cover():
4   while there exists an uncovered matched pair do
5      $\ell =$  smallest index such that  $(\alpha_\ell, \alpha'_\ell)$  is not covered
6     if  $|\alpha'_\ell - \alpha_\ell| \leq w_{\max}$  then
7        $p = \ell$ 
8        $mx = \alpha'_\ell$ 
9       while  $|(\alpha_\ell, \alpha'_{p+1})| \leq w_{\max}$  do
10         $p \leftarrow p + 1$ 
11        if  $\alpha'_p \geq mx$  then
12           $mx \leftarrow \alpha'_p$ 
13      add  $(\alpha_\ell, mx)$  to  $W_1$ 
14    else
15       $q_1 = \max_{q \geq \ell, |\alpha_\ell - \alpha_q| \leq w_{\max}} q$ 
16       $q_2 = \max_{q \geq \ell, |\alpha'_q - \alpha'_\ell| \leq w_{\max}} q$ 
17       $p = \min\{q_1, q_2\}$ 
18      add  $(s[(\alpha_\ell, \alpha_p)(\alpha'_p, \alpha'_\ell)])$  to  $Q_1$ 

```

However, some of the windows may be smaller than w_{\min} and some of the super-windows may have a sub-window whose size is smaller than w_{\min} . In the second step, we remove all windows with size less than $w_{\min} + \gamma$ from W_1 and all the super-windows of Q_1 such that either of their sub-windows have length smaller than $w_{\min} + \gamma$. The reason that we use $w_{\min} + \gamma$ instead of w_{\min} is to keep the length of the windows/super-windows large enough to make sure that at least one of the windows/super-windows in $W \cup Q$ is totally within each window/super-window.

More precisely, let the new set of windows and super-windows for s be W_2 and Q_2 . We add each super-window q of Q_1 with the property that both of its sub-windows have length at least $w_{\min} + \gamma$ to Q_2 . We ignore all super-windows of Q_1 whose sub-windows have length smaller than $w_{\min} + \gamma$. If for a super-window q of Q_1 only one of its sub-windows has length at least $w_{\min} + \gamma$ we operate as follows: let $s[a, b]$ be this sub-window. We find c, d where c and d are respectively the smallest and the largest indices such that $a < c < d < b$, each one of s_c, s_d corresponds to a matched pair, and $\text{tw}(s_c), \text{tw}(s_d)$ are also in $s[a, b]$. Indeed, $s[c, d]$ is the smallest substring of $s[a, b]$ that includes all the matched pairs in $s[a, b]$ with the property that their twins are also in $s[a, b]$. For brevity, we call $s[c, d]$, the *covering window* of $s[a, b]$. If no such c, d exists, or $d - c + 1 < w_{\min} + \gamma$, we also ignore this sub-window. Otherwise, we add $s[c, d]$ as a window to S_2 .

Similarly, for each window w of W_1 we add it to W_2 only if its length is at least $w_{\min} + \gamma$. Below, we show that W_2 and Q_2 only miss a small number of matched pairs. We begin by stating Lemma 6 which proves a bound on the number of super-windows of Q_1 and then use it to also prove a bound on the number of windows of W_1 .

► **Lemma 6.** $|Q_1| \leq 4n/w_{\max}$.

Proof. We prove that since our algorithm constructs Q_1 from s , then we have $|s| \geq (|Q_1| + 1)w_{\max}/2$. Since the original trees have at most n nodes, we have $|s| \leq 2n$ which in turn implies $|Q_1| \leq 4n/w_{\max}$.

For a super-window $q \in Q_1$, define $f(q)$ to be the number of super-windows of Q_1 that are nested under q (including q itself). We claim that for any super-window $q = [(a, b), (b', a')]$ of Q_1 , we have $a' - a \geq (f(q) + 1)w_{\max}/2$. To show this, we use induction on the value of $f(q)$. The base case is when $f(q) = 1$ which is trivial: since the twin pair a, a' is covered by a super-window, we imply that $a' - a \geq w_{\max}$. Therefore, for $q = [(a, b), (b', a')]$ we have

$$a' - a \geq w_{\max} = \frac{(f(q) + 1)w_{\max}}{2}. \quad (2)$$

Now, assume that for some $k > 1$, the argument is correct for all super-windows q^* such that $f(q^*) < k$. We use this to prove the hypothesis for some super-window $q = [(a, b), (b', a')]$ where $f(q) = k$. Let $\langle q_1, q_2, \dots, q_l \rangle$ be the set of all super-windows of Q_1 which have the following properties: (i) Each super-window q_i is nested under q . (ii) For each super-window q_i , there is no super-window $q' \in Q_1$ such that q_i is nested under q' and q' is nested under q . In other words, each super-window of Q_1 which is nested under q is either one of q_i 's or nested under one of q_i 's. This implies that $f(q) = \sum f(q_i) + 1$.

$$\left(\dots \overbrace{(\dots)}^{q_1} \dots \overbrace{(\dots)}^{q_2} \dots \dots \overbrace{(\dots)}^{q_l} \dots \right)$$

$$a \quad a_1 \quad a'_1 \quad a_2 \quad a'_2 \quad \dots \quad a_l \quad a'_l \quad a'$$

For each super-window q_i , we denote the positions of its starting character and its ending character in s by a_i and a'_i . By induction hypothesis, we know that for every $1 \leq i \leq l$, we have

$$a'_i - a_i \geq \frac{(f(q_i) + 1)w_{\max}}{2}. \quad (3)$$

On the other hand, we have

$$f(q) = 1 + \sum_{1 \leq i \leq l} f(q_i). \quad (4)$$

If $l \geq 2$, Inequality (3) and Equation (4) directly imply Inequality (2) since we have

$$\begin{aligned} a' - a &\geq \sum_{1 \leq i \leq l} a'_i - a_i \\ &\geq \sum_{1 \leq i \leq l} \frac{f(q_i) + 1}{2} w_{\max} \\ &= \frac{f(q) + (l - 1)}{2} w_{\max} && \text{Inequality (4)} \\ &\geq \frac{f(q) + 1}{2} w_{\max} && l \geq 2 \end{aligned}$$

Therefore, it remains to prove Inequality (2) for $l = 1$. In this case, we make use of the following observation: Due to Algorithm 1, $\max\{a_1 - a, a' - a'_1\} \geq w_{\max}$ holds since otherwise Algorithm 1 would extend q by bringing in some character from q_1 . Moreover, by induction hypothesis we have

$$a'_1 - a_1 \geq \left(\frac{f(q) - 1 + 1}{2} \right) w_{\max}.$$

and therefore,

$$a' - a \geq \left(\frac{f(q) - 1 + 1}{2} \right) w_{\max} + w_{\max} \geq \left(\frac{f(q) + 1}{2} \right) w_{\max}. \quad \blacktriangleleft$$

Recall that we make Q_2 from Q_1 by only including super-windows whose sub-windows are of size at least $w_{\min} + \gamma$. Moreover, if for a super-window, only one of its sub-windows has a length at least $w_{\min} + \gamma$, we find the covering window of that sub-window³ and add it to W_2 as an independent window only if its length is at least $w_{\min} + \gamma$. To find the amount of matched pairs we lose in this process, define k_1, k_2 , and k_3 as follows:

- k_1 : the number of super-windows whose sub-windows have length smaller than $w_{\min} + \gamma$.
- k_2 : the number of super-windows such that one of their sub-windows has length smaller than $w_{\min} + \gamma$ and the covering window of the other sub-window has length at least $w_{\min} + \gamma$.
- k_3 : the number of super-windows such that one of their sub-windows has length smaller than $w_{\min} + \gamma$, but the covering window of the other sub-window has also length smaller than $w_{\min} + \gamma$.

The total number of matched pairs of s we miss by refining sub-windows is at most

$$\begin{aligned} 2(w_{\min} + \gamma)k_1 + 2(w_{\min} + \gamma)k_2 + (2(w_{\min} + \gamma) + (w_{\min} + \gamma))k_3 &\leq 3(w_{\min} + \gamma)(k_3 + k_2 + k_1) \\ &\leq 12n(w_{\min} + \gamma)/w_{\max}. \end{aligned}$$

Below, we also bound the number of matched pairs of s we miss by refining the windows.

► **Lemma 7.** $|W_1| \leq 13n/w_{\max}$.

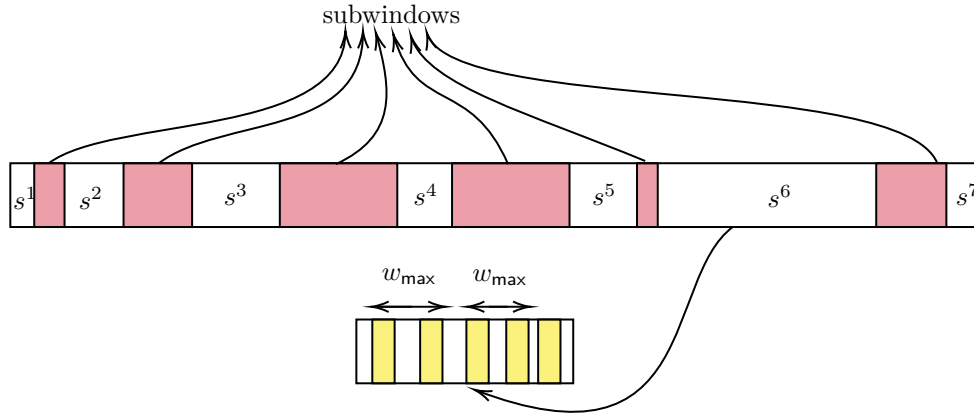
Proof. Let $w_1 = s[a, b]$ and $w_2 = s[c, d]$ be two consecutive windows of W_1 . Then, either $d - a \geq w_{\max}$ holds or w_1 and w_2 are separated by a sub-window of some super-window in Q_1 since otherwise Algorithm 1 never adds w_1 to W_1 . Lemma 6 implies that the total number of sub-windows of Q_1 is at most $2|Q_1| = 8n/w_{\max}$. If we remove these sub-windows from s , what remains from s is a collection $\langle s^1, s^2, \dots, s^\beta \rangle$ of fragments of s where $\beta \leq 1 + 8n/w_{\max}$. On the other hand, since for each pair w_1, w_2 of consecutive windows in s^i , we know that the distance between the starting index of w_1 to the ending index of w_2 is at least w_{\max} , we conclude that the number of windows that are within s^i is at most $1 + \lfloor 2|s^i|/w_{\max} \rfloor$. Summing this over all the fragments indicates that the total number of windows is at most

$$\begin{aligned} \sum_{1 \leq i \leq \beta} 1 + \lfloor 2|s^i|/w_{\max} \rfloor &\leq \beta + \sum_{1 \leq i \leq \beta} \lfloor 2|s^i|/w_{\max} \rfloor \\ &\leq 1 + 8n/w_{\max} + \sum_{1 \leq i \leq \beta} \lfloor 2|s^i|/w_{\max} \rfloor \\ &\leq 1 + 8n/w_{\max} + \lfloor 4n/w_{\max} \rfloor \\ &\leq 1 + 12n/w_{\max} \\ &\leq 13n/w_{\max}. \end{aligned}$$

See Figure 5 for an example. ◀

Lemma 7 implies that the total number of matched pairs of s we lose by eliminating short windows is at most $13n(w_{\min} + \gamma)/w_{\max}$. This in addition to the matched pairs we lose for super-windows amounts to a loss of at most $25n(w_{\min} + \gamma)/w_{\max}$.

³ The smallest substring of a sub-window that includes all the matched pairs with the property that their twins are also in that sub-window.



■ **Figure 5** In this figure, s is divided by 6 sub-windows that belong to 3 super-windows. If we remove these sub-windows from s , what remains of s is a collection of 7 fragments. Since for each pair w_1, w_2 of consecutive windows in s^i we know that the distance between the starting index of w_1 to the ending index of w_2 is at least w_{\max} , the number of windows that are within s_i is at most $1 + \lfloor 2|s_i|/w_{\max} \rfloor$. For example, the number of windows in s^6 is at most $\lfloor \frac{2|s^6|}{w_{\max}} \rfloor + 1$.

▶ **Corollary 8** (of Lemmas 7 and 6). *Windows and super-windows of W_2 and Q_2 cover all but at most $25n(w_{\min} + \gamma)/w_{\max}$ matched pairs of s .*

We make the windows and super-windows of \bar{s} from W_2 and Q_2 . More precisely, for each window of W_2 we put a window in \bar{W}_2 that covers the corresponding matched pairs in \bar{s} and for each super-window of Q_2 we put a super-window in \bar{Q}_2 that covers the corresponding matched pairs in \bar{s} . It follows from Corollary 8 that a window-compatible transformation with respect to windows and super-windows of W_2 , Q_2 , \bar{W}_2 , and \bar{Q}_2 has an additive error of at most $25n(w_{\min} + \gamma)/w_{\max}$.

▶ **Lemma 9.** *There exists a window-compatible transformation between the two trees T and \bar{T} with respect to windows of W_2 , Q_2 , \bar{W}_2 , and \bar{Q}_2 that has a cost of at most $|\text{opt}| + 25n(w_{\min} + \gamma)/w_{\max}$.*

Proof. The proof follows directly from Corollary 8. ◀

In the third step, we refine the windows and super-windows of W_2 , Q_2 , \bar{W}_2 , and \bar{Q}_2 . We make W_3 , Q_3 , \bar{W}_3 , and \bar{Q}_3 in the following way: Let w and \bar{w} be a pair of windows of W_2 and \bar{W}_2 that cover the same set of matched pairs in the two strings. If $w_{\min} + \gamma \leq |\bar{w}| \leq \bar{w}_{\max}$ then we put w into W_3 and \bar{w} into \bar{W}_3 . Similarly, let q and \bar{q} be a pair of super-windows of Q_2 and \bar{Q}_2 that cover the same set of matched characters in the two strings. If both sub-windows of \bar{q} have a length in range $[w_{\min} + \gamma, \bar{w}_{\max}]$ then we put q in Q_3 and \bar{q} in \bar{Q}_3 . If this condition only holds for one of the sub-windows, we operate as follows: let $\bar{s}[a, b]$ be this sub-window and let $s[a, b]$ be its corresponding sub-window in s . In addition, let $s[c, d]$ and $\bar{s}[\bar{c}, \bar{d}]$ be the covering windows of $s[a, b]$ and $s[a, b]$. We add $s[c, d]$ and $\bar{s}[\bar{c}, \bar{d}]$ to W_3 and \bar{W}_3 as a pair of independent windows, if and only if the size of both of them is at least $w_{\min} + \gamma$.

We prove in Lemma 10 that a window-compatible transformation by the windows and super-windows of W_3 , Q_3 , \bar{W}_3 , and \bar{Q}_3 has a cost of at most $|\text{opt}| + 54n(w_{\min} + \gamma)/w_{\max} + 6nw_{\max}/\bar{w}_{\max}$.

► **Lemma 10.** *There exists a window-compatible transformation between the two trees T and \bar{T} with respect to windows of W_3 , Q_3 , \bar{W}_3 , and \bar{Q}_3 that has a cost of at most $|\text{opt}| + 54n(w_{\min} + \gamma)/w_{\max} + 6nw_{\max}/\bar{w}_{\max}$.*

Proof. Let w and \bar{w} be two matched windows in W_2 and \bar{W}_2 such that $|\bar{w}| < w_{\min} + \gamma$. Since $|w| \geq w_{\min} + \gamma$, a trivial upper bound on the number of matched pairs that belong to w and \bar{w} is $\min(|w|, |\bar{w}|) \leq w_{\min} + \gamma$. On the other hand, by Lemmas 6 and 7, the total number of windows in W_2 is at most $17n/w_{\max}$, that is $13n/w_{\max}$ windows in W_1 plus at most $4n/w_{\max}$ possible windows made from sub-windows. Therefore, the total number of matched pairs we lose by removing the windows with size smaller than w_{\min} from \bar{W}_2 along with their corresponding windows from W_2 is $17n(w_{\min} + \gamma)/w_{\max}$. Also, the number of windows in \bar{W}_2 with length more than \bar{w}_{\max} is at most $2n/\bar{w}_{\max}$. Each one of these windows contains at most w_{\max} matched pairs, since the size of their corresponding window in W_2 is at most w_{\max} . Hence, the total number of matched pairs we lose by removing the windows with size more than \bar{w}_{\max} from \bar{W}_2 along with their corresponding windows from W_2 is at most $2nw_{\max}/\bar{w}_{\max}$.

For super-windows, we consider two cases. First, consider the super-windows in \bar{Q}_2 whose sub-windows have length smaller than $w_{\min} + \gamma$. Since these super-windows are not included in \bar{Q}_3 , for each one of these super-windows, we lose $2(w_{\min} + \gamma)$ many matched pairs. Also, consider the super-windows in \bar{Q}_2 such that one of their sub-windows has length less than $w_{\min} + \gamma$ and the other sub-window has length in $[w_{\min} + \gamma, w_{\max}]$. For each one of these super-windows, we check the covering windows of the larger sub-window and its counterpart in s , and add them to W_3 and \bar{W}_3 if and only if their size is at least $w_{\min} + \gamma$. Hence, for each one of these super-windows we lose at most $2(w_{\min} + \gamma) + (w_{\min} + \gamma) = 3(w_{\min} + \gamma)$ number of matched pairs. Since the total number of super-windows is at most $4n/w_{\max}$, this results in the total loss of at most $12n(w_{\min} + \gamma)/w_{\max}$. Also, the number of super-windows in \bar{W}_2 such that the size of one of their sub-windows is more than \bar{w}_{\max} is at most $2n/\bar{w}_{\max}$. Each one of these super-windows contains at most $2w_{\max}$ matched pairs. Hence, the total number of matched pairs we lose by removing these super-windows is at most $4nw_{\max}/\bar{w}_{\max}$. Therefore, we conclude that the process of constructing W_3 , Q_3 , \bar{W}_3 , and \bar{Q}_3 eliminates at most $29n(w_{\min} + \gamma)/w_{\max} + 6nw_{\max}/\bar{w}_{\max}$ matched pairs. This, together with the error we calculated for W_2 and Q_2 means that the optimal window-compatible transformation between T and \bar{T} with respect to W_3 , Q_3 , \bar{W}_3 , and \bar{Q}_3 has a cost of at most $|\text{opt}| + (54n(w_{\min} + \gamma)/w_{\max}) + (6nw_{\max}/\bar{w}_{\max})$. ◀

Finally, in the last step, we construct W_4 , Q_4 , \bar{W}_4 , and \bar{Q}_4 from W_3 , Q_3 , \bar{W}_3 , and \bar{Q}_3 in a way that not only a window-compatible solution based on W_4 , Q_4 , \bar{W}_4 , and \bar{Q}_4 preserves the approximation factor, but also we have $W_4 \subseteq W$, $Q_4 \subseteq Q$, $\bar{W}_4 \subseteq \bar{W}$, and $\bar{Q}_4 \subseteq \bar{Q}$. We explain below how W_4 and Q_4 are made from W_3 and Q_3 . The windows and super-windows of \bar{W}_4 and \bar{Q}_4 are made similarly from \bar{W}_3 and \bar{Q}_3 .

Let $w = s[a, b]$ be a window of W_3 for which we would like to add an equivalent window of W_3 into W_4 . By the construction of the windows of W and Q we know that for some i we have $a \in [\mathcal{G}_i, \mathcal{G}_{i+1})$. Let c be the rightmost index in $[\mathcal{G}_i, \mathcal{G}_{i+1})$ such that s_c is an opening parenthesis. By definition, inequalities $a \leq c$ and $c - a \leq \gamma$ hold. Also, since $s[c, b]$ is a valid window, by the way we construct W , we know W includes a window of form $s[c, d]$, such that $d \leq b$, and furthermore,

$$\frac{b - c + 1}{d - c + 1} \leq 1 + \epsilon_{\text{win}}. \quad (5)$$

We add $w' = s[c, d]$ to W_4 as the representative of w . Since $c - a \leq \gamma$, we have $|w'| \geq (|w| - \gamma)/(1 + \epsilon_{\text{win}})$. Therefore, the number of matched characters we lose is at most

$$2(|w| - |w'|) \leq 2|w|\epsilon_{\text{win}} + 2\gamma. \quad (6)$$

Choosing a proper super-window for each super-window of Q_3 is a more challenging task. Let $q = s[(a, b), (b', a')]$ be a super-window of Q_3 , and fix i, j in a way that $a \in [\mathcal{G}_i, \mathcal{G}_{i+1})$, $a' \in [\mathcal{G}_j, \mathcal{G}_{j+1})$. We choose two super-windows and show that at least one of these super-windows is eligible to be in Q_4 as the representative of q .

Let c be the rightmost opening parenthesis in $[\mathcal{G}_i, \mathcal{G}_{i+1})$ such that $\text{tw}(c) \in [b', a']$ and let $c' = \text{tw}(c)$. Since $s[(c, b)(b', c')]$ is a valid super-window, by the way we construct Q , Q includes a super-window of form $s[(c, d), (d', c')]$ such that

$$\frac{b - c + 1}{d - c + 1} \leq 1 + \epsilon_{\text{win}} \quad \text{and} \quad \frac{c' - b' + 1}{c' - d' + 1} \leq 1 + \epsilon_{\text{win}}.$$

Since $c - a \leq \gamma$, we have $b - a + 1 \leq (d - c + 1)(1 + \epsilon_{\text{win}}) + \gamma$. If the same holds for the right sub-windows, i.e., $a' - c' \leq \gamma$, then we have

$$a' - b' + 1 \leq (c' - d' + 1)(1 + \epsilon_{\text{win}}) + \gamma, \quad (7)$$

and therefore we can add $s[(c, d), (d', c')]$ to Q_4 as the representative of q . However, it might be the case that $a' - c' > \gamma$ and therefore Inequality (7) does not hold. If $a' - c' > \gamma$, let f' be the left-most closing parenthesis of $[\mathcal{G}_j, \mathcal{G}_{j+1}]$ such that $f = \text{tw}(f') \in [a, b]$. Since $f' \in [\mathcal{G}_j, \mathcal{G}_{j+1}]$ we have $a' - f' \leq \gamma$ and therefore, $f' > c'$ and consequently $f < c$. In fact, twin pair (f, f') encloses twin pair (c, c') . Since $s[(f, b), (b', f')]$ is a valid super-window, by the way we construct Q , we know Q includes a super-window of form $s[(f, g), (g', f')]$ such that

$$\frac{b - f + 1}{g - f + 1} \leq 1 + \epsilon_{\text{win}} \quad \text{and} \quad \frac{f' - b' + 1}{f' - g' + 1} \leq 1 + \epsilon_{\text{win}}.$$

Note that since $a' - f' \leq \gamma$, we have

$$a' - b' + 1 \leq (g' - f' + 1)(1 + \epsilon_{\text{win}}) + \gamma.$$

Finally, since $f < c$ we have $f - a < \gamma$ and so

$$b - a + 1 \leq (g - f + 1)(1 + \epsilon_{\text{win}}) + \gamma.$$

Therefore, we can add $s[(f, g)(g', f')]$ to Q_4 as the representative of q . The number of matched pairs we lose in both sub-windows here is $2(|q|\epsilon_{\text{win}} + 2\gamma)$.

In conclusion, after construing W_4, \overline{W}_4, Q_4 , and \overline{Q}_4 , the number of matched pairs in a window or super-window w that are covered in W_3, \overline{W}_3, Q_3 , and \overline{Q}_3 and might not be covered by its representative in W_4, \overline{W}_4, Q_4 , and \overline{Q}_4 is at most

$$\sum_{w \in W_4 \cup \overline{W}_4} (2|w|\epsilon_{\text{win}} + 2\gamma) + \sum_{q \in Q_4 \cup \overline{Q}_4} (2|q|\epsilon_{\text{win}} + 4\gamma) \leq 8n\epsilon_{\text{win}} + 136\gamma/w_{\text{max}} \quad (8)$$

In Inequality (8) we use the facts that the total number of windows and super-windows for each string is at most $17n/w_{\text{max}}$ and the total length of the windows/super-windows in W_4, \overline{W}_4, Q_4 , and \overline{Q}_4 sums up to $4n$. This together with the error we calculated for previous steps implies that the cost of the optimal window-compatible solution with respect to W_4, \overline{W}_4, Q_4 , and \overline{Q}_4 is bounded by

$$|\text{opt}| + (8n\epsilon_{\text{win}}) + (190n\gamma/w_{\text{max}}) + (54nw_{\text{min}}/w_{\text{max}}) + (6nw_{\text{max}}/\overline{w}_{\text{max}}).$$

This completes the proof of Lemma 2.

► Remark 11. Before we conclude this section, we would like to make a note: the proof of Lemma 2 implies that there exists a window-compatible solution with the desired guarantee such that each window of s is transformed into a window of \bar{s} and each super-window of s is transformed into a window of \bar{s} . Therefore, we do not need to worry about solutions that transform a window into a super-window or vice-versa (although this would not be a computational barrier).

References

- 1 Shyan Akmal and Ce Jin. Faster algorithms for bounded tree edit distance. In *ICALP*, pages 12:1–12:15. Springer, 2021.
- 2 Tatsuya Akutsu, Daiji Fukagawa, and Atsuhiko Takasu. Approximating tree edit distance through string edit distance. *Algorithmica*, 57(2):325–348, 2010.
- 3 Alexandr Andoni and Negev Shekel Nosatzki. Edit distance in near-linear time: it’s a constant factor. In *FOCS*, pages 990–1001. IEEE, 2020.
- 4 Philip Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1):217–239, 2005.
- 5 Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi HajiAghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and MapReduce. In *SODA*, pages 1170–1189. SIAM, 2018.
- 6 Mahdi Boroujeni, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, and Saeed Seddighin. $1+\epsilon$ approximation of tree edit distance in quadratic time. In *STOC*, pages 709–720. ACM, 2019.
- 7 Mahdi Boroujeni, Masoud Seddighin, and Saeed Seddighin. Improved algorithms for edit distance and lcs: beyond worst case. In *SODA*, pages 1601–1620. SIAM, 2020.
- 8 Joshua Brakensiek and Aviad Rubinfeld. Constant-factor approximation of near-linear edit distance in near-linear time. In *STOC*, pages 685–698. ACM, 2020.
- 9 Karl Bringmann, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Tree edit distance cannot be computed in strongly subcubic time (unless APSP can). In *SODA*, pages 1190–1206. SIAM, 2018.
- 10 Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *VLDB*, pages 141–152. VLDB Endowment, 2003.
- 11 Horst Bunke and Kim Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3):255–259, 1998.
- 12 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucky, and Michael Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *FOCS*, pages 979–990. IEEE, 2018.
- 13 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael Saks. Approximating edit distance within constant factor in truly sub-quadratic time. *Journal of the ACM (JACM)*, 67(6):1–22, 2020.
- 14 Sudarshan S. Chawathe. Comparing hierarchical data in external memory. In *VLDB*, pages 90–101. Morgan Kaufmann Publishers Inc., 1999.
- 15 Erik D Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. In *ICALP*, pages 146–157. Springer, 2007.
- 16 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM (JACM)*, 57(1):4:1–4:33, November 2009.
- 17 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- 18 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing (SIOMP)*, 13(2):338–355, 1984.
- 19 Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *ESA*, pages 91–102. Springer, 1998.

- 20 Michal Koucký and Michael Saks. Constant factor approximations to edit distance on far input pairs in nearly linear time. In *STOC*, pages 699–712. ACM, 2020.
- 21 Gad M Landau, Eugene W Myers, and Jeanette P Schmidt. Incremental string comparison. *SIAM Journal on Computing (SICOMP)*, 27(2):557–582, 1998.
- 22 Xiao Mao. Breaking the cubic barrier for (unweighted) tree edit distance. In *FOCS*. IEEE, 2021.
- 23 Aviad Rubinfeld, Saeed Seddighin, Zhao Song, and Xiaorui Sun. Approximation algorithms for lcs and lis with truly improved running times. In *FOCS*, pages 1121–1145. IEEE, 2019.
- 24 Stanley M Selkow. The tree-to-tree editing problem. *Information processing letters*, 6(6):184–186, 1977.
- 25 Bruce A. Shapiro and Kaizhong Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Bioinformatics*, 6(4):309–318, 1990.
- 26 Kuo Chung Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, July 1979.
- 27 Hélène Touzet. A linear tree edit distance algorithm for similar ordered trees. In *CPM*, pages 334–345. Springer, 2005.
- 28 Michael S Waterman. *Introduction to computational biology: maps, sequences and genomes*. CRC Press, 1995.
- 29 Kaizhong Zhang and Dennis E. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing (SICOMP)*, 18(6):1245–1262, 1989.