Edit Distance between Unrooted Trees in Cubic Time

Bartłomiej Dudek
Institute of Computer Science, University of Wrocław, Poland
bartlomiej.dudek@cs.uni.wroc.pl

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland
gawry@cs.uni.wroc.pl

Abstract

Edit distance between trees is a natural generalization of the classical edit distance between strings, in which the allowed elementary operations are contraction, uncontraction and relabeling of an edge. Demaine et al. [ACM Trans. on Algorithms, 6(1), 2009] showed how to compute the edit distance between rooted trees on n nodes in $O(n^3)$ time. However, generalizing their method to unrooted trees seems quite problematic, and the most efficient known solution remains to be the previous $O(n^3 \log n)$ time algorithm by Klein [ESA 1998]. Given the lack of progress on improving this complexity, it might appear that unrooted trees are simply more difficult than rooted trees. We show that this is, in fact, not the case, and edit distance between unrooted trees on n nodes can be computed in $O(n^3)$ time. A significantly faster solution is unlikely to exist, as Bringmann et al. [SODA 2018] proved that the complexity of computing the edit distance between rooted trees cannot be decreased to $O(n^{3-\varepsilon})$ unless some popular conjecture fails, and the lower bound easily extends to unrooted trees. We also show that for two unrooted trees of size m and n, where $m \leq n$, our algorithm can be modified to run in $O(nm^2(1 + \log \frac{n}{m}))$. This, again, matches the complexity achieved by Demaine et al. for rooted trees, who also showed that this is optimal if we restrict ourselves to the so-called decomposition algorithms.

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

Keywords and phrases tree edit distance, dynamic programming, heavy light decomposition

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.45

Related Version A full version of the paper is available at [14], `https://arxiv.org/abs/1804.10186`.

1 Introduction

Computing the edit distance between two strings [30] is the most well-known example of dynamic programming. Thanks to the new fine-grained complexity paradigm, we know that this simple approach is essentially the best possible [1, 5], so the problem appears to be solved from the theoretical perspective. However, in many real-life applications we would like to operate on more complicated structures than strings. As a prime example, while primary structure of RNA can be seen as a string, computational biology is often interested in comparing also secondary structures. Second structure of RNA can be modeled as an ordered tree [17, 26], so we would like to generalize computing the edit distance between strings to computing the edit distance between ordered trees.

Tai [29] defined the edit distance between two ordered trees as the minimum total cost of a sequence of elementary operations that transform one tree into the other. For unrooted

45th International Colloquium on Automata, Languages, and Programming (ICALP 2018).
Editors: Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella;
Article No. 45; pp. 45:1–45:14


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

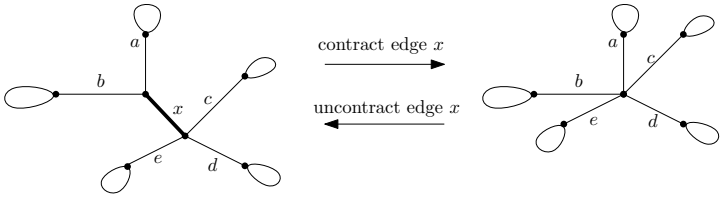

Figure 1 Contraction and uncontraction of the edge with label x costs $c_{del}(x) = c_{ins}(x)$.

trees, which are the focus of this paper, the trees are edge-labeled, and we have three elementary operations: contraction, uncontraction and relabeling of an edge. We think that the trees are embedded in the plane, i.e., there is a cyclic order on the neighbors of every node that is preserved by the contraction/uncontraction. See Figure 1. The cost of an operation depends on the label(s) of the edge(s): $c_{del}(\tau)$, $c_{ins}(\tau)$, $c_{match}(\tau_1, \tau_2)$, respectively. We assume that every operation has the same cost as its reverse counterpart: $c_{del}(\tau) = c_{ins}(\tau)$, $c_{match}(\tau_1, \tau_2) = c_{match}(\tau_2, \tau_1)$, and each edge participates in at most one elementary operation.

Computing the edit distance between trees is used as a measure of similarity in multiple contexts. The most obvious, given that some biological structures resemble trees, is computational biology [26]. Others include comparing XML data [10, 11, 16], programming languages [18]. Others, less obvious, include computer vision [6, 20, 22, 25], character recognition [24], automatic grading [3], and answer extraction [31]. See also the survey by Bille [7].

Tai [29] introduced the edit distance between rooted node-labeled trees on n nodes and designed an $O(n^6)$ algorithm. Zhang and Shasha [27] improved the time complexity to $O(n^4)$ by designing a recursive formula, which reduces computing the edit distance between two trees to computing the edit distance between two smaller trees. Then, Klein [21] considered the more general problem of computing the edit distance between unrooted edge-labeled trees and further improved the complexity to $O(n^3 \log n)$ using essentially the same formula, but applying it more carefully to restrict the number of different trees that appear in the whole process. This high-level idea of using the recursive formula can be formalized using the notion of decomposition strategy algorithms as done by Dulucq and Touzet [15]. Finally, Demaine et al. [13] further improved the complexity for rooted node-labeled trees to $O(n^3)$. For trees of different sizes m and n, where $m \leq n$, their algorithm runs in $O(nm^2(1 + \log \frac{n}{m}))$ time. At a very high level, the gist of their improvement was to apply the heavy path decomposition to both trees, while in Klein's algorithm only one tree is decomposed. This requires some care, as switching from being guided by the heavy path decomposition of the first tree to the second tree cannot be done too often.

Although Demaine et al. [13] showed that their algorithm is optimal among all decomposition strategies, it is not clear that any algorithm must be based on such a strategy. Nevertheless, there has been no progress on beating the best known $O(n^3)$ time worst-case bound for exact tree edit distance. Pawlik and Augsten [23] presented an experimental comparison of the known algorithms. Aratsu et al. [4], Akutsu et al. [2], and Ivkin [19] designed approximation algorithms. Only very recently a convincing explanation for the lack of progress on improving this worst-case complexity has been found by Bringmann et al. [9], who showed that a significant improvement on the cubic time complexity for rooted node-labeled trees is rather unlikely: an $O(n^{3-\varepsilon}$ algorithm for computing the edit distance between rooted trees on n nodes implies an $O(n^{3-\varepsilon})$ algorithm for APSP (assuming alphabet of size $\Theta(n)$) and an $O(n^{k(1-\varepsilon)})$ algorithm for Max-Weight k-Clique (assuming alphabet of sufficiently large but constant size).

Thus, the complexity of computing the edit distance between rooted trees seems well-understood by now. However, in multiple important applications, the trees are, in fact, unrooted. For example, Sebastian et al. [25] use unrooted trees to recognize shapes (in a paper with over 700 citations). Unfortunately, while the almost 20 years old algorithm presented by Klein works for unrooted trees in $O(n^3 \log n)$ time, it is not clear how to translate Demaine et al.'s improvement to the unrooted case. In fact, even if one of the trees is a rooted full binary tree and the other is a simple caterpillar, their approach appears to use $O(n^4)$ time, and it is not clear how to modify it. Given the lack of further progress, it might seem that unrooted trees are simply more difficult than rooted trees.

Our contribution. We present a new algorithm for computing the edit distance between unrooted trees which runs in $O(n^3)$ time and $O(n^2)$ space. For the case of trees of possibly different sizes n and m where $m \leq n$, it runs in $O(nm^2(1 + \log \frac{n}{m}))$ time and $O(nm)$ space. This matches the complexity of Demaine et al.'s algorithm for the rooted case and improves Klein's algorithm for the unrooted case. By a simple reduction, unrooted trees are as difficult as rooted trees, so our algorithm is optimal among all decomposition algorithms [13], and significantly faster approach is unlikely to exist unless some popular conjecture fails [9].

Our starting point is dynamic programming using the recursive formula of Zhang and Shasha, similarly as done by Klein and Demaine et al., but instead of presenting the computation in a top-down order, we prefer to work bottom-up. This gives us more control and allows us to be more precise about the details of the implementation. In the simpler $O(n^3 \log \log n)$ version of the algorithm, we apply the heavy path decomposition to both trees. As long as the first tree is sufficiently big, we proceed similarly as Klein, that is, look at its heavy path decomposition. However, if the first tree is small (roughly speaking) we consider the heavy path decomposition of the second tree and design a new divide and conquer strategy that is applied on every heavy path separately.

In the full version of the paper [14] we further improve the complexity to $O(n^3)$. Instead of a global parameter we modify the divide and conquer strategy so that the larger the first tree is the sooner the strategy terminates and switches to another approach. A careful analysis of such a modification leads to $O(nm^2(1 + \log^2 \frac{n}{m})) = O(n^3)$ running time. Then, we shave one $\log \frac{n}{m}$ by making the divide and conquer sensitive to the sizes of the subtrees attached to the heavy path instead of its length, that is, making some nodes more important than the other, reminiscing the so-called telescoping trick [8, 12]. All the improvements applied together decrease the overall complexity to $O(nm^2(1 + \log \frac{n}{m}))$, thus matching the running time of the algorithm by Demaine et al. for rooted trees [13].

▶ **Theorem 1.** *Edit distance between unrooted trees of size n and m, where $m \leq n$, can be computed in $O(nm^2(1 + \log \frac{n}{m})) = O(n^3)$ time and $O(nm)$ space.*

A straightforward reduction shows that computing edit distance between unrooted trees is at least as difficult as computing edit distance between rooted trees. Thus, invoking the lowerbound of Demaine et al. [13] we obtain that our algorithm is optimal if we restrict ourselves to the so-called decomposition algorithms, and by the result of Bringmann et al. [9] a significantly faster $O(n^{3-\varepsilon})$ algorithm is not possible assuming some popular conjecture.

Roadmap. In Section 2 we introduce the notation and the recursive formula that are then used to present Klein's algorithm adapted for the rooted case. Next, in Section 3 we return to the unrooted case, introduce new notation and transform both input trees by adding some auxiliary edges. Then, in Section 4 we present our new $O(n^3 \log \log n)$ algorithm for the

unrooted case which already improves the state-of-the-art Klein's algorithm and is essential for understanding our main $O(n^3)$ algorithm described in the full version of the paper. Both algorithms are described in a bottom-up fashion. In the simpler $O(n^3 \log\log n)$ version we first assume that one of the trees is a caterpillar and then generalize to arbitrary trees. In the more complicated $O(n^3)$ algorithm we start with an even more restricted case of one tree being a caterpillar and the other a rooted full binary tree. When analyzing both algorithms we only bound the total number of considered subproblems. As explained in the full version, this can be translated into an implementation with the same running time.

2 Preliminaries

We are given two unrooted trees T_1, T_2 with every edge labeled by an element of Σ and a cyclic order on the neighbors of every node. For every label $\alpha \in \Sigma$, we know the cost $c_{del}(\alpha) = c_{ins}(\alpha)$ of contracting or uncontracting of an edge with label α. For every $\alpha, \beta \in \Sigma$, $c_{match}(\alpha, \beta) = c_{match}(\beta, \alpha)$ is the cost of changing the label of an edge from α to β. All costs are non-negative and each edge can participate in at most one operation. Edit distance between T_1 and T_2 is defined as the minimum total cost of a sequence of the above operations transforming T_1 to T_2. Equivalently, it is the minimum cost of transforming both the trees to a common tree using only contracting and relabeling operations, as each operation has the same cost as its undo-counterpart. Note that for unrooted trees, edit distance is the minimum edit distance over all possible rootings of T_1 and T_2, where a rooting is uniquely determined by choice of the root and the leftmost edge from the root.

We first assume, that both trees are of equal size $n = |T_1| = |T_2|$, but later we will also address the case when one of them is significantly larger than the other. We start with the case when both trees are rooted, which is essential for the understanding of the unrooted case. Then, every node has its children ordered left-to-right. We also assume that both (rooted) trees are binary, as we can add $O(n)$ edges with a fresh label that costs 0 to contract and ∞ to relabel.

Naming convention. We use a similar naming convention as in [13]. We call main left and right edges of a (rooted) tree respectively the leftmost and rightmost edge from the root. For a given rooted tree T with at least 2 nodes, let r_T denote the right main edge of T and R_T denote the rooted subtree of T that is under (not including) r_T. By $T - r_T$ we denote a tree obtained from T by contracting edge r_T and by $T - R_T$ a tree obtained from T by contracting edge r_T and all edges from its subtree R_T. Thus the tree T consists of R_T, the edge r_T and edges $(T - R_T)$. l_T and L_T are defined analogously and T^v denotes subtree of T rooted at v. See Figure 2(a) and (b).

We define a pruned subtree of a tree T to be the tree obtained from T by a sequence of contractions of the left or right main edge. Note that every pruned subtree is uniquely represented by the pair of its left and right main edges. It also corresponds to an interval on the Euler tour of the tree started in the root when we remove from the interval each edge that occurs once. Thus we can completely represent a pruned subtree in $O(1)$ space by storing two edges. We can preprocess all the $O(n^2)$ pruned subtrees T' of a tree T to be able to obtain trees $R_{T'}, L_{T'}, T' - l_{T'}, T' - r_{T'}$ and edges $r_{T'}, l_{T'}$ in $O(1)$ time.

Dynamic programming. Zhang and Shasha [27] introduced the following recursive formula for computing the edit distance between two rooted trees:

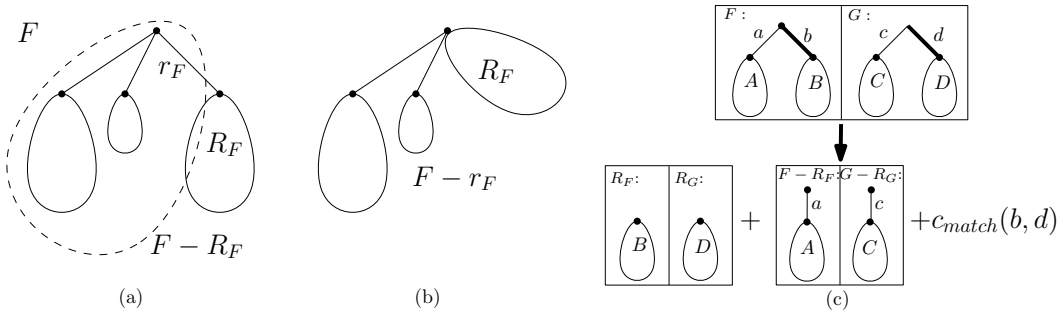

Figure 2 (a) Tree F with both r_F and R_G contracted. (b) F with its right main edge contracted. (c) When both right main edges are not contracted we obtain two independent problems.

▶ **Lemma 2.** *Let $\delta(F, G)$ be the edit distance between two pruned subtrees F and G of respectively T_1 and T_2. Then:*

- $\delta(\emptyset, \emptyset) = 0$
- $$\delta(F, G) = \min \begin{cases} \delta(F - r_F, G) + c_{del}(r_F) & \text{if } F \neq \emptyset \\ \delta(F, G - r_G) + c_{del}(r_G) & \text{if } G \neq \emptyset \\ \delta(R_F, R_G) + \delta(F - R_F, G - R_G) + c_{match}(r_F, r_G) & \text{if } F, G \neq \emptyset \end{cases}$$

The above recurrence also holds if we contract or match the left main edge.

It contracts the right main edge in one of the two trees or matches the right main edges of the two trees. In the latter case, we get two independent subproblems (R_F, R_G) and $(F - R_F, G - R_G)$ that must be transformed to equal trees. See Figure 2(c) for an illustration of this case.

To estimate time complexity of the algorithm, we only count different pairs (F, G) for which $\delta(F, G)$ is computed. Each such value is computed at most once and stored. Note that F is always a pruned subtree of T_1, while G is a pruned subtree of T_2, thus there are $O(n^4)$ possible pairs (F, G). In the worst case, all such pairs might be considered. The formula from Lemma 2 can be evaluated in constant time, and any previously computed value can be retrieved in constant time from a four-dimensional table.

The above algorithm always contracts or relabels the right main edge. A more deliberate choice of direction (whether to choose the left or right main edge) will lead to a different behavior of the algorithm which in turn might result in a smaller total number of considered pairs (F, G). Such a family of algorithms is called decomposition algorithms. When analyzing the time complexity of such an algorithm, we assume that any already computed $\delta(F, G)$ can be retrieved in constant time. If our goal is to compute significantly fewer than $O(n^4)$ subproblems, we cannot afford to allocate the four-dimensional table anymore. An obvious solution is to store the already computed values in a hash table, but this requires randomization. In the full version [14] we explain how to carefully arrange the order of the computation and store the partial results as to obtain deterministic algorithms with the same running time.

While the formula from Lemma 2 suggests a top-down strategy, we phrase the algorithms in a bottom-up perspective, which allows us to present the details of the computation more precisely. The aim of all the algorithms is to compute $\delta(T_1, T_2)$ knowing only $\delta(\emptyset, \cdot)$ and $\delta(\cdot, \emptyset)$, as the costs of contraction of an arbitrary pruned subtree are precomputed.

Algorithm 1 Klein's algorithm.

```
1: for each heavy path H in T_1 in the bottom-up order do
2:     let v_1, v_2, ..., v_{|H|} = H
3:     for i = |H| - 1, ..., 0 do
           ▷ avoiding the heavy child:
4:         ComputeFrom(δ(T_1^{v_i}, ·), δ(T_1^{v_{i+1}}, ·))
```

Klein's $O(n^3 \log n)$ algorithm. Klein's algorithm [21] uses heavy path decomposition [28] of T_1. The root is called light and every node calls its child with the largest subtree (and the leftmost in case of ties) heavy and all other children light. An edge is heavy if it leads to the heavy child.

While applying the dynamic formula from Lemma 2, Klein's algorithm uses a strategy that we call "avoiding the heavy child" in T_1. It chooses the direction (either left or right) in such a way that the edge leading to the heavy child of the root is contracted or relabeled as late as possible. Observe that contracting the main edge not leading to the heavy child of the root of a pruned subtree T, does not change the heavy child of the root of T, as its subtree is still the largest. Note that Klein's strategy does not depend on the considered pruned subtree of T_2.

Even though Klein uses top-down view to describe his algorithm, we find it more convenient to implement the computations in bottom-up order. Therefore the algorithm processes heavy paths of T_1 in the bottom-up order as shown in Algorithm 1. Consider a heavy path H with nodes $v_1, v_2, \ldots, v_{|H|}$ where v_1 is the closest node to the root and $v_{|H|}$ is a leaf. By $\delta(T_1^v, \cdot)$ we denote a table of $O(n^2)$ distances between tree T_1^v and all pruned subtrees of T_2. The algorithm considers all nodes on H also bottom-up. It starts from $\delta(T_1^{v_{|H|}}, \cdot) = \delta(\emptyset, \cdot)$, which is precomputed, and then iteratively computes $\delta(T_1^{v_i}, \cdot)$ from $\delta(T_1^{v_{i+1}}, \cdot)$ for decreasing values of i. We denote such a step by ComputeFrom subroutine. Note that in every step the strategy avoiding the heavy child always chooses the same direction (recall that the tree is binary) and visits altogether at most $O(n)$ pruned subtrees of T_1. Also when actually implementing the ComputeFrom step we proceed bottom-up. That is, suppose we have already computed $\delta(T_1^{v_{i+1}}, \cdot)$ and that v_{i+1} is the left child of v_i. Then the strategy avoiding the heavy child says R that is chooses first the right main edge to consider. We compute $\delta(T_1^{v_i}, \cdot)$ as follows. First we consider the tree $T_1^{v_{i+1}} \cup \{\{v_i, v_{i+1}\}\}$ (we call this uncontracting the heavy edge), next $T_1^{v_{i+1}} \cup \{\{v_i, v_{i+1}\}, \{v_i, w\}\}$ if exists a light child w of v_i and then uncontract the subsequent edges of T_1^w. This guarantees that while computing $\delta(F, G)$ the subtrees $F - r_F$ and $F - R_F$ have been already processed. Pruned subtrees of T_2 are also considered in the order of increasing sizes. Clearly, as argued for Zhang and Shasha's algorithm, the algorithm visits $O(n^2)$ pruned subtrees of T_2, so we need to bound the number of pruned subtrees of T_1.

▶ **Observation 3.** Consider an arbitrary tree T. Suppose that strategy avoiding the heavy child in T says R for a pruned subtree F. Then $F - R_F$ is also obtained by a sequence of contractions of the main edge according to the strategy.

The observation implies that in order to count the relevant intervals of T_1 we can only consider the trees obtained by contraction of the main edge according to the strategy, and trees of the form L_F and R_F. Note that the only trees of the form L_F or R_F that are not obtained in this way are rooted at a light node so will be counted separately for another heavy path.

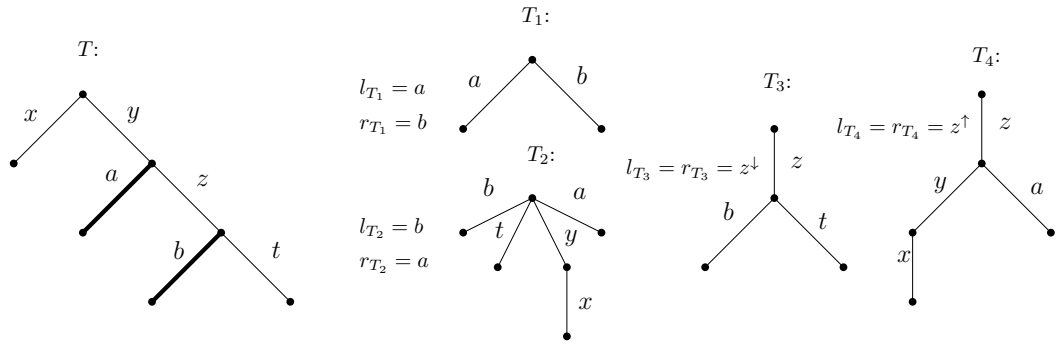

Figure 3 Every pruned subtree is uniquely represented by its left and right main edges or a dart.

We denote $\mathsf{apex}(F)$ to be the top node of the heavy path containing the lowest common ancestor of all endpoints of edges of F. In other words, $\mathsf{apex}(F)$ is the lowest light ancestor of all edges of F. Now grouping all the visited pruned subtrees by their apex-es we bound their total number:

▶ **Observation 4.** For an arbitrary tree T, there is $\sum_{v\text{: light node in } T} |T^v|$ pruned subtrees of T visited while applying strategy avoiding the heavy child of T.

Let the light-depth $\mathsf{ldepth}(u)$ of a node u be the number of light nodes that are ancestors of u (node is also an ancestor of itself). Because $\mathsf{ldepth}(u) \leq \log(n) + 1$, we obtain:

$$\sum_{v\text{: light node in } T_1} |T_1^v| = \sum_{v\text{: node in } T_1} \mathsf{ldepth}(v) \in O(n \log n) \tag{1}$$

Recalling that there are $O(n^2)$ relevant intervals of T_2 we conclude that Klein's algorithm visits $O(n^3 \log n)$ subproblems. As we assume the constant time memoization, it runs in $O(n^3 \log n)$ time.

3 Back to Unrooted Case

Recall that edit distance between two unrooted trees T_1 and T_2 is the minimum edit distance between T_1 and T_2 over all possible rootings of them, where rooting is determined by the root of the tree and its the left main edge. As Klein [21] mentioned, it is enough to choose an arbitrary rooting in one of the trees and try all possible rootings of the other to find an optimal setting. Observe, that we can treat the Euler tour of T_2 as a cyclic string and represent every pruned subtree of T_2 as an interval of it, for all possible rootings of T_2. Thus Klein's algorithm works in $O(n^3 \log n)$ time also for the edit distance between unrooted trees. Before we present our faster algorithm for this case, we need to introduce some new definitions. Recall, that even in the unrooted case, we first arbitrarily root both trees and the initial rooting remains unchanged throughout the algorithm.

Darts. We replace every edge e with two darts corresponding to two ways of traversing the edge, either down $e^{\downarrow}$ or up the tree $e^{\uparrow}$ (with respect to the fixed rooting). Subtree of a dart $\mathsf{subtree}((u, v))$ is defined as the subtree rooted at node v, when u is its parent. Note that $e^{\uparrow}$ and $e^{\downarrow}$ belong neither to $\mathsf{subtree}(e^{\uparrow})$ nor to $\mathsf{subtree}(e^{\downarrow})$. Every pruned subtree of (unrooted) tree is uniquely represented by its left and right main edges or a dart (if there is one edge from the root). See Figure 3.

Auxiliary edges for rootings. We observed that every rooting of T_2 corresponds to a subrange of a cyclic Euler tour E_{T_2}, but later it will be convenient to represent every rooting as a subtree of a dart. For this purpose, we add new edges labeled with a fresh label $\# \notin \Sigma$ which will be used only to denote a rooting. Setting $c_{del}(\#) = 0$ and $c_{match}(\#, \cdot) = \infty$ we force that these edges are only contracted. For every node v we add new edges alternating with the original ones. Thus in total, there are $2(n-1)$ edges added. Using these new edges we can compute edit distance between the unrooted trees from the values of $\delta(T_1, \mathsf{subtree}(d))$ for all darts d in T_2. Thus, our aim is to fill the table Δ where $\Delta[u, d] := \delta(T_1^u, \mathsf{subtree}(d))$.

Auxiliary edges to bound the degrees. As the last step, again we add $O(n)$ edges with appropriate costs as to ensure that the degree of every node is at most 3. Observe that the cost of the optimal solution for the modified trees is the same as for the initial ones and having a sequence of operations for the modified trees, we can easily obtain an optimal sequence for the original instance of the problem.

4 $O(n^3 \log\log n)$ Algorithm for Unrooted Case

After initial modifications both trees are binary and the algorithm needs to fill the table Δ where $\Delta[u, d] := \delta(T_1^u, \mathsf{subtree}(d))$ for all nodes $u \in T_1$ and darts $d \in T_2$. We first run Demaine et al.'s algorithm operating on labels on edges instead of nodes which computes $\delta(T_1^u, T_2^v)$ for all nodes $u \in T_1$ and $v \in T_2$ in $O(n^3)$ time and stores them in $\Delta[u, e_v^{\downarrow}]$ where $e_v^{\downarrow}$ is the dart to v from its parent. Now we need to fill the remaining fields $\Delta[u, e^{\uparrow}]$ for all darts $e^{\uparrow}$ up the tree T_2.

This is the main difficulty in the unrooted case, in which we need to handle many big subtrees which are significantly different from each other. Our approach is to successively reduce different subproblems to smaller ones, in a way that there are fewer subproblems to consider in the next step. We use divide and conquer paradigm, in which there is more and more sharing after every step.

In the beginning, we call each node of T_1 and T_2 light or heavy as in the Klein's algorithm and all the time the notion is with respect to the initial rootings. Similarly, the notion of traversing an edge up or down the tree is always with respect to the rooting. Recall that we denote $\mathsf{apex}(T)$ as the top node on the heavy path containing the lowest common ancestor of all edges of T. We first fix a global value b, which will be determined exactly later. On a high level, from the top-down perspective, the algorithm uses the following strategy to compute $\delta(F, G)$: if $|T_1^{\mathsf{apex}(F)}| > n/b$, then avoid the heavy child in F, and otherwise apply a new strategy based only on G and T_2.

Considering it bottom-up, the algorithm first fills values of $\Delta[u, e^{\uparrow}]$ for all nodes u such that $|T_1^{\mathsf{apex}(u)}| \leq n/b$ and all darts up T_2. For the remaining fields of Δ, it uses strategy avoiding the heavy child in T_1. As in the Klein's algorithm, in this phase, the algorithm needs to process heavy paths of T_1 in the bottom-up order. Note that for each light node v such that $|T_1^v| > n/b$ holds $\mathsf{ldepth}(v) < \log b + 1$. Thus there are $O(n^3 \log b)$ subproblems visited in total in this phase.

For the other phase note that there are $O(n^2/b)$ relevant subtrees in T_1, and now we need to carefully design and analyze the new strategy for T_2. It will be easier to think, that in this phase the algorithm needs to compute $\Delta[u, e^{\uparrow}]$ for all darts $e^{\uparrow}$ up T_2 and all nodes $u \in T_1$ such that $|T_1^u| \leq n/b$, call them interesting. Clearly, all subproblems in which there is a switch to the strategy based on T_2 are of this form.

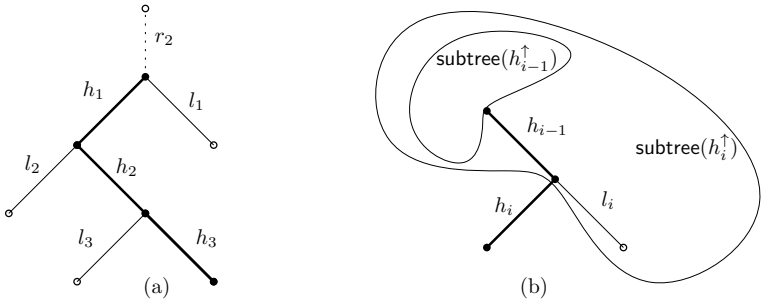

Figure 4 (a) A heavy path H with edge r_2 (dotted) denoting the rooting of T_2. (b) To compute $\delta(*, \mathsf{subtree}(h_i^{\uparrow}))$ we use $\delta(*, \mathsf{subtree}(h_{i-1}^{\uparrow}))$, first uncontract the edge h_{i-1} and then l_i (if exists).

As now the strategy will be more complex than before, we first describe it for the case when T_2 is a caterpillar: a heavy path with possibly single nodes connected to it. This example is already difficult in the unrooted case and will require divide and conquer approach to handle all the possible rootings of T_2 at once. Next, we will slightly modify the approach to handle arbitrary trees T_2.

4.1 Caterpillar T_2

Now we consider the case when T_2 is a heavy path H with possibly single nodes connected to it. Let h_i denote (heavy) edges on H, $r_2 = h_0$ be the edge denoting the initial rooting of T_2 and (if exists) l_i be the light edge connected to the i-th node on H. See Figure 4(a) for an example.

In the first step we compute values of $\delta(*, \mathsf{subtree}(h_i^{\uparrow}))$ for all heavy edges h_i, where $*$ denotes all pruned subtrees of T_1 of size at most n/b. The strategy is to avoid the parent, that is to contract the edge leading to the parent as late as possible. See Figure 4(b).

More precisely, in the beginning, we already know $\delta(*, \mathsf{subtree}(h_0^{\uparrow}))$, because it is the cost of contraction of the whole pruned subtree of T_1 (which is precomputed), as $h_0 = r_2$ and $\mathsf{subtree}(h_0^{\uparrow}) = \emptyset$. Then, having values of $\delta(*, \mathsf{subtree}(h_{i-1}^{\uparrow}))$ we compute $\delta(*, \mathsf{subtree}(h_i^{\uparrow}))$ by uncontracting first h_{i-1} and then l_i if it exists. It is an extension of the ComputeFrom subroutine, but now we do not have subtrees T^x and T^y, where x is the parent of y, but two edges h_i and h_{i-1} with a common endpoint. Note that in this step all uncontractions are from the same direction.

There are $O(n)$ pruned subtrees of T_2 obtained by uncontractions of a main edge according to the strategy, starting from the empty subtree. Now we need to show that the algorithm did not consider any other pruned subtree of T_2. Suppose it uncontracted the left main edge. Then $G - L_G \in \{\emptyset, G - l_G\}$, depending on whether l_G was the heavy edge leading to the parent or not. Also $L_G \in \{\emptyset, G - l_G\}$, so in both cases, all the obtained pruned subtrees are among the $O(n)$ described above. Finally, as there are $O(n^2/b)$ pruned subtrees of T_1, in total we computed and stored the edit distance of $O(n^3/b)$ subproblems. Now, using the computed values we fill $\Delta[u, h_i^{\uparrow}]$ for all interesting nodes $u \in T_1$ and heavy edges $h_i \in T_2$. Thus, later on, we do not have to consider the pruned subtrees of the form $\delta(L_F, L_G)$ or $\delta(R_F, R_G)$ as their values are already stored in Δ, because either one of them is empty or they are of the form $\delta(T_1^v, \mathsf{subtree}(dh))$ for an interesting node $u \in T_1$ and a dart dh from a heavy edge in T_2. We only have not computed values $\Delta[u, l^{\uparrow}]$ for darts from light edges up the tree, but in this phase of the algorithm, they never appear in $\delta(L_F, L_G)$ or $\delta(R_F, R_G)$ subproblem. However, we need to compute these values because they correspond to some rootings of T_2, so we will consider them in the following paragraph.

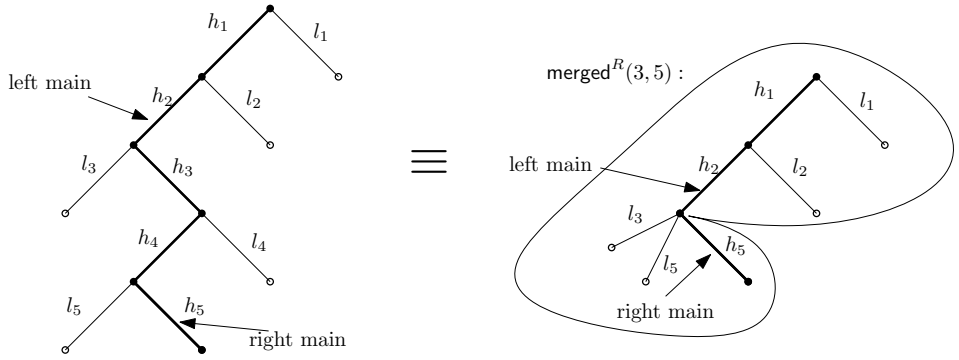

Figure 5 Pruned subtree $\mathsf{merged}^R(3,5)$ has the left main edge h_2 and right h_5.

Algorithm 2 Computes input tables needed for processing a heavy path H

1: **function** ProcessHeavyPath($\delta(*, \mathsf{subtree}(h_0^\uparrow))$)
2: **for** $i = 1..|H|$ **do**
▷ avoiding the parent:
3: ComputeFrom($\delta(*, \mathsf{subtree}(h_i^\uparrow)), \delta(*, \mathsf{subtree}(h_{i-1}^\uparrow))$)
4: fill $\Delta[u, h_i^\uparrow]$ for all interesting nodes u

▷ repeatedly uncontracting the left main edge:
5: ComputeFrom($\delta(*, \mathsf{merged}^R(1, |H|)), \delta(*, \mathsf{subtree}(h_0^\uparrow))$)
▷ repeatedly uncontracting the right main edge:
6: ComputeFrom($\delta(*, \mathsf{merged}^L(1, |H|)), \delta(*, \mathsf{subtree}(h_0^\uparrow))$)

7: Group($1, |H|, \mathsf{Data}(1, |H|)$)

Darts from light nodes up the tree. From now on, our algorithm processes heavy paths of T_2 one-by-one. In particular, in this subsection, we process the only heavy path H of T_2. Thus, unless explicitly stated otherwise all the notion is relative to the current heavy path H. First, we define $\mathsf{merged}^R(A, B)$ as the pruned subtree obtained by contraction of edges between the A-th and B-th node on H or to the right of H:

▶ **Definition 5.** Let H be a heavy path and A and B ($A \leq B$) denote indices of two nodes on H. Then $\mathsf{merged}^R(A, B)$ is a tree with the left main edge h_{A-1} and the right main edge h_B. $\mathsf{merged}^L(A, B)$ is a tree with the left main edge h_B and the right main edge h_{A-1}.

See Figure 5. Note that $\mathsf{subtree}(l_A^\uparrow)$ is either $\mathsf{merged}^R(A, A)$ or $\mathsf{merged}^L(A, A)$, depending on which side of H is l_A.

As explained earlier, in the beginning the algorithm computes $\delta(*, \mathsf{subtree}(h_i^\uparrow))$ for all heavy edges on H. Additionally, it calculates $\delta(*, \mathsf{merged}^L(1, |H|))$ and $\delta(*, \mathsf{merged}^R(1, |H|))$ from $\delta(*, \mathsf{subtree}(h_0^\uparrow))$ by repeatedly uncontracting respectively the right and left main edge. See Algorithm 2 for the summary of the whole preprocessing. Then it calls a recursive procedure Group($1, |H|, \mathsf{Data}(1, |H|)$). The final goal of this call is to fill $\Delta[u, l_i^\uparrow]$ for all light edges l_i connected to the heavy path H.

Group($A, B, \mathsf{Data}(A, B)$) is a procedure which considers an interval $[A, B]$ of indices on H given tables of values $\delta(*, \mathsf{subtree}(h_{A-1}^\uparrow)), \delta(*, \mathsf{subtree}(h_B^\downarrow)), \delta(*, \mathsf{merged}^L(A, B))$ and $\delta(*, \mathsf{merged}^R(A, B))$, which we denote as $\mathsf{Data}(A, B)$. Intuitively, $\mathsf{Data}(A, B)$ contains in-

Algorithm 3 Fills $\Delta[u, l_i^{\uparrow}]$ for light edges l_i connected to the heavy path H with $i \in [A, B]$.

```
1:  function GROUP(A, B, Data(A, B))
2:      if A = B then
3:          if there is a light edge l_A connected to H then
4:              fill Δ[u, l_A^↑] for interesting nodes u ∈ T_1
5:          return
6:      M := ⌊(A + B)/2⌋
7:      for i = (B − 1)..M do
            ▷ avoiding the heavy child:
8:          COMPUTEFROM(δ(∗, subtree(h_i^↓)), δ(∗, subtree(h_{i+1}^↓)))
        ▷ repeatedly uncontracting the right main edge:
9:      COMPUTEFROM(δ(∗, merged^R(A, M)), {δ(∗, merged^R(A, B)); δ(∗, subtree(h_{A−1}^↑))})
        ▷ repeatedly uncontracting the left main edge:
10:     COMPUTEFROM(δ(∗, merged^L(A, M)), {δ(∗, merged^L(A, B)); δ(∗, subtree(h_{A−1}^↑))})

11:     GROUP(A, M, Data(A, M))
12:     symmetric computations for interval [M + 1, B]
13:     GROUP(M + 1, B, Data(M + 1, B))
```

formation about subtrees "outside" the considered interval $[A, B]$ which are relevant during intermediate computations. Then, the procedure calls itself recursively for shorter intervals until it holds that $A = B$ when $\delta(*, \mathsf{merged}^L(A, A))$ or $\delta(*, \mathsf{merged}^R(A, A))$ contains the fields of $\Delta[u, l_A^{\uparrow}]$ for all interesting nodes u and then the recurrence stops.

In more detail, for an interval $[A, B]$, the procedure computes $\mathsf{Data}(A, M)$ and $\mathsf{Data}(M + 1, B)$ for $M = \lfloor \frac{A+B}{2} \rfloor$ and calls itself recursively for the smaller intervals. Note that for $\mathsf{Data}(A, M)$ it needs to compute tables $\delta(*, G)$ for trees $G = \mathsf{merged}^R(A, M), \mathsf{merged}^L(A, M)$ or $\mathsf{subtree}(h_M^{\downarrow})$ and can reuse table $\delta(*, \mathsf{subtree}(h_{A-1}^{\uparrow}))$ which is a part of $\mathsf{Data}(A, B)$. Similarly for interval $[M + 1, B]$. See Algorithm 3.

To analyze the complexity of the GROUP procedure, first note that in every step of the loop in line 7, it considers a constant number of pruned subtrees from T_2, so in total there are $O(B - M)$ of them. After this loop, we have $\delta(*, \mathsf{subtree}(h_M^{\downarrow}))$ computed.

The call of COMPUTEFROM in line 9 needs more input than the call in line 8, even though the strategy is always uncontracting the right main edge. Note that if the dynamic program only tried contracting the right main edge, it would be possible to compute $\delta(*, \mathsf{merged}^R(A, M))$ only from $\delta(*, \mathsf{merged}^R(A, B))$. However, it is not the case when the algorithm also matches edges. The first case when r_G is a light edge ($r_G = l_X$ for some value of X) is not problematic, because then $R_G = \emptyset$ and $G - R_G = G - r_G$, so this pruned subtree is already visited. Although, if r_G is a heavy edge then $R_G = \mathsf{subtree}(r_G^{\downarrow})$ and $G - R_G$ is a pruned subtree, which has not been considered yet. Observe that in this situation the pruned subtree can be obtained from $\mathsf{subtree}(h_{A-1}^{\uparrow})$ by a sequence of $O(B - A)$ contractions of the right main edge, so we need it as a separate input to the COMPUTEFROM subroutine. A similar reasoning applies to the edges to the left of H in line 10 and to the computations for interval $[M + 1, B]$.

To sum up, one call of GROUP(A, B) (not including recursive calls) visits $O(B - A)$ pruned subtrees of T_2. As we start from an interval of length $|H|$ and in every recursive call its length is roughly halved, the procedure considers in total $O(|H| \log |H|) = O(n \log n)$ pruned subtrees of T_2.

4.2 Arbitrary Tree T_2

Now we describe, how to modify the above algorithm to process not only a caterpillar, but an arbitrary tree T_2. In this case, there can be non-empty subtrees connected to the main heavy path.

Note that for an arbitrary heavy path H inside T_2, the ProcessHeavyPath procedure only needs to know $\delta(*, \mathsf{subtree}(h_0^{\uparrow}))$ to be able to compute all the remaining input parameters in $\mathsf{Data}(1, |H|)$, because $\delta(*, \mathsf{subtree}(h_{|H|}^{\downarrow})) = \delta(*, \emptyset)$ is precomputed. In the beginning, the algorithm calls $\textsc{ProcessHeavyPath}_{H^0}(\delta(*, \emptyset))$, where H^0 is the heavy path of T_2 containing the root of T_2. The only place we need to change inside the Group procedure to handle arbitrary trees T_2 is to not only fill $\Delta[u, l_A^{\uparrow}]$ in line 4 of Algorithm 3, but also recursively call $\textsc{ProcessHeavyPath}_{H'}(\delta(*, \mathsf{subtree}(l_A^{\uparrow})))$ where H' is the heavy path connected to the A-th node of the considered heavy path. As we pointed earlier, $\mathsf{subtree}(l_A^{\uparrow})$ is either $\mathsf{merged}^R(A, A)$ or $\mathsf{merged}^L(A, A)$, depending on which side of H is l_A. Now observe, that each subsequent pruned subtree that appears in the recursive formula is already visited and processed:

▶ **Observation 6.** In the modified Group procedure, during the call of ComputeFrom subroutine in line 9 of Algorithm 3, all the intermediate pruned subtrees of T_2 are obtained by a sequence of uncontractions of the right main edge from the root either from $\mathsf{merged}^R(A, B)$ or $\mathsf{subtree}(h_{A-1}^{\uparrow})$. A similar property holds for the other three calls of ComputeFrom in lines 10 and 12.

What changes in the analysis of the procedure is that now there are not $O(|H| \log |H|)$ pruned subtrees of T_2 but $O(|T_2^v| \log |H|) = O(|T_2^v| \log n)$, where v is the top node of H. In other words, the heavy path H itself might be short, but there might be big subtrees connected to it. However, every subtree connected to H is completely contracted (edge-by-edge) a constant number of times on every level of recursion of Group procedure and thus the bound.

Recall that top node of every heavy path is light, so using equation (1) we bound the overall number of subtrees of T_2 considered during this part of the algorithm:

$$\sum_{v\text{: top node of a heavy path in } T_2} |T_2^v| \cdot \log n = \sum_{v\text{: light node in } T_2} |T_2^v| \cdot \log n \in O(n \log^2 n)$$

4.3 Final Analysis

To conclude, the above algorithm computes $\Delta[u, e^{\uparrow}]$ for all nodes $u \in T_1$ such that $|T_1^u| \leq n/b$ and all darts up the tree T_2 by considering $O(n \log^2 n)$ pruned subtrees of T_2 and $O(n^2/b)$ of T_1. At the beginning of Section 4 we described the second phase of the algorithm, which avoids the heavy child in T_1 and fills the remaining fields of Δ considering $O(n \log b)$ pruned subtrees of T_1 and $O(n^2)$ of T_2. Thus, during the two phases, the whole algorithm visits $O(n^3 \frac{\log^2 n}{b} + n^3 \log b)$ subproblems. Setting $b = \log^2 n$ we obtain the overall complexity $O(n^3 \log \log n)$.

References

1 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *48th STOC*, pages 375–388, 2016.

2 Tatsuya Akutsu, Daiji Fukagawa, and Atsuhiro Takasu. Approximating tree edit distance through string edit distance. *Algorithmica*, 57(2):325–348, 2010.

3 Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of DFA constructions. In *23rd IJCAI*, pages 1976–1982, 2013.

4 Taku Aratsu, Kouichi Hirata, and Tetsuji Kuboyama. Approximating tree edit distance through string edit distance for binary tree codes. *Fundam. Inf.*, 101(3):157–171, 2010.

5 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *47th STOC*, pages 51–58, 2015.

6 J. Bellando and R. Kothari. Region-based modeling and tree edit distance as a basis for gesture recognition. In *10th ICIAP*, pages 698–703, 1999.

7 Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

8 Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theor. Comput. Sci.*, 11(3):303–320, 1980.

9 Karl Bringmann, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Tree edit distance cannot be computed in strongly subcubic time (unless APSP can). In *29th SODA*, 2018.

10 Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *29th VLDB*, pages 141–152, 2003.

11 Sudarshan S. Chawathe. Comparing hierarchical data in external memory. In *25th VLDB*, pages 90–101, 1999.

12 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *36th STOC*, pages 91–100, 2004.

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

14 Bartłomiej Dudek and Paweł Gawrychowski. Edit distance between unrooted trees in cubic time. *CoRR*, abs/1804.10186, 2018. `arXiv:1804.10186`.

15 Serge Dulucq and Hélène Touzet. Decomposition algorithms for the tree edit distance problem. *J. Discrete Algorithms*, 3(2-4):448–471, 2005.

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

17 Matthias Höchsmann, Thomas Töller, Robert Giegerich, and Stefan Kurtz. Local similarity in RNA secondary structures. In *2nd CSB*, pages 159–168, 2003.

18 Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, Jan. 1982.

19 Egor Ivkin. Approximating tree edit distance through string edit distance for binary tree codes. B.Sc. thesis, Charles University in Prague, 2012.

20 Philip Klein, Srikanta Tirthapura, Daniel Sharvit, and Ben Kimia. A tree-edit-distance algorithm for comparing simple, closed shapes. In *11th SODA*, pages 696–704, 2000.

21 Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *6th ESA*, pages 91–102, 1998.

22 Philip N. Klein, Thomas B. Sebastian, and Benjamin B. Kimia. Shape matching using edit-distance: An implementation. In *12th SODA*, pages 781–790, 2001.

23 Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, 2015.

24 Juan Ramón Rico-Juan and Luisa Micó. Comparison of aesa and laesa search algorithms using string and tree-edit-distances. *Pattern Recogn. Lett.*, 24(9-10):1417–1426, jun 2003.

25 T. B. Sebastian, P. N. Klein, and B. B. Kimia. Recognition of shapes by editing their shock graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(5):550–571, May 2004.

26 B. A. Shapiro and K. Z. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Comput. Appl. Biosci.*, 6(4):309–318, Oct. 1990.

27 Dennis Shasha and Kaizhong Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms*, 11(4):581–621, dec 1990.

28 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

29 Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

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

31 Xuchen Yao, Benjamin Van Durme, Chris Callison-Burch, and Peter Clark. Answer extraction as sequence tagging with tree edit distance. In *HLT-NAACL*, pages 858–867, 2013.