Edit Distance between Unrooted Trees in Cubic Time

Bartłomiej Dudek

Praca magisterska
napisana pod kierunkiem
dr. Pawła Gawrychowskiego

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Wrocław 2017

Streszczenie

Odległość edycyjna pomiędzy drzewami jest naturalnym uogólnieniem klasycznego problemu odległości edycyjnej pomiędzy słowami. Znajduje ona zastosowanie przede wszystkim w biologii obliczeniowej, a dokładniej w porównywaniu struktur RNA, a także w innych dziedzinach, na przykład w przetwarzaniu obrazu. Efektywne obliczanie odległości edycyjnej pomiędzy słowami jest prostym ćwiczeniem z algorytmów, ale dla drzew staje się to dużo trudniejsze. Demaine et al. [ACM Trans. on Algorithms, 6(1), 2009] skonstruowali algorytm znajdujący odległość edycyjną pomiędzy ukorzenionymi drzewami na n wierzchołkach w czasie $O(n^3)$. Uogólnienie ich podejścia dla drzew nieukorzenionych wydaje się jednak dość problematyczne, i najlepszym znanym rozwiązaniem dla tej wersji pozostaje dużo wcześniejszy algorytm Kleina [ESA 1998] o wyższej złożoności $O(n^3 \log n)$. Jako że drzewa nieukorzenione wydają się dużo bardziej skomplikowane niż ukorzenione, można by podejrzewać, że złożoność problemu dla tych pierwszych po prostu musi być większa. W tej pracy pokazujemy, że wcale tak nie jest, bowiem odległość edycyjna pomiędzy drzewami nieukorzenionymi na n wierzchołkach może być wyliczona w czasie $O(n^3)$. Bringmann et al. [arXiv:1703.08940, 2017] pokazali, że istnienie algorytmu o złożoności $O(n^{3-\varepsilon})$ dla drzew ukorzenionych nie jest możliwe zakładając jedną z popularnych hipotez używanych do pokazywania tego typu stwierdzeń. Ich wynik łatwo przenieść na drzewa nieukorzenione, więc istnienie algorytmu istotnie szybszego niż $O(n^3)$ jest mało prawdopodobne.

Uzyskanie złożoności $O(n^3)$ wymaga wielu nowych pomysłów w porównaniu do poprzedniego algorytmu działającego w takim czasie dla drzew ukorzenionych. Podobnie jak Demaine et al., zaczynamy od podziału obu drzew na ciężkie i lekkie ścieżki, jednak przedstawiamy ich podejście w inny, łatwiejszy do uogólnienia sposób. Początkowo, jedno z drzew dzielimy na górną i dolną część i każdą z nich przetwarzamy w inny sposób, przy czym dla każdej ścieżki w górnej części stosujemy nową technikę dziel i zwyciężaj. Uważna analiza tych pomysłów już wystarcza do rozwiązania o złożoności $O(n^3 \log \log n)$. Aby je usprawnić, uzależniamy podział drzewa na części od rozważanej ciężkiej ścieżki w drugim z drzew, a także zmieniamy podejście dziel i zwyciężaj, by brało pod uwagę rozmiary poddrzew przyczepionych do rozważanej ścieżki, a nie tylko jej długość. Na koniec trzeba dokładnie przeanalizować sumaryczny czas działania całego algorytmu.

Pokazujemy także jak zmodyfikować nasz algorytm tak, aby działał w czasie $O(nm^2(1 + \log \frac{n}{m}))$ dla dwóch drzew nieukorzenionych o rozmiarach m i n. Demaine et el. udowodnili, że taka złożoność jest optymalna dla drzew ukorzenionych jeśli ograniczymy się do dość ogólnej klasy algorytmów dekompozycji. Dodatkowo pokazujemy, że cały algorytm można zaimplementować w pamięci $O(nm)$.

Abstract

Edit distance between trees is a natural generalization of the classical edit distance between strings. Its prime applications include computational biology, more specifically comparing RNA secondary structures, and (less obviously) computer vision. While computing the edit distance between strings is a basic undergraduate exercise in algorithms, designing an efficient algorithm for trees is more challenging. After a series of improvements, 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, for which the most efficient known solution remains to be the previous $O(n^3 \log n)$ time algorithm by Klein [ESA 1998]. Due to 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. [arXiv:1703.08940, 2017] 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.

Achieving $O(n^3)$ time for unrooted trees requires a number of new ideas compared to the previous algorithm with the same complexity for rooted trees. The starting point is heavy path decomposition of both trees as used by Demaine et al., but we provide a slightly different formulation of their method, which seems more convenient for further improvements. Then, we additionally separate one of the trees in a bottom and top part, which are processed differently. For the top part, we develop a new divide and conquer procedure that is applied on every path separately. This is already enough to achieve $O(n^3 \log \log n)$ complexity. To further improve on that, we make the partition into bottom and top part dependent on the currently considered heavy path of the other tree, and additionally change the divide and conquer procedure as to be more sensitive to the sizes of the subtrees attached to the heavy path instead of the length of the path. This needs to be then carefully analyzed over all the heavy paths.

We also show that for two unrooted trees of size m and n, where $m \leqslant n$, 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. Additionally, we show that our algorithm can be implemented in only $O(nm)$ space.

Contents

1 Introduction

Computing the edit distance between two strings [27] is probably the most well-known example of dynamic programming. Even though it has many real-life applications, in some cases we want to operate on more complicated structures than strings. A natural generalization is computing the edit distance between two trees introduced by Tai [26].

The edit distance between ordered trees is defined as the minimum total cost of a sequence of elementary operations that transform one tree into the other. The exact definition depends on whether the trees are rooted or unrooted. For unrooted 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. If the trees are rooted, it is more natural to consider node-labeled trees. The operations are then defined similarly, except that the cost now depends on the label of the participating node. While for rooted trees working with node- or edge-labeled trees is just a matter of preference, for unrooted trees, it is not so clear what does it mean to contract or uncontract a node. Therefore, we will work with edge-labeled trees only.

To see that the above definition is a natural generalization of the standard edit distance between strings, convert a string of length n into a rooted path on $n + 1$ nodes, where the label of the edge connecting the i-th and $(i + 1)$-th node is the same as the i-th character of the original string.

Computing the edit distance between trees is used as a measure of similarity between trees in multiple contexts. The most obvious, given that some biological structures resemble trees, is computational biology [23]. Others include comparing XML data [9, 10, 14], programming languages [15]. Others, less obvious, include computer vision [4,17,19,22], character recognition [21], automatic grading [2], and answer extraction [28]. See also the survey by Bille [6].

Tai [26] introduced the edit distance between rooted node-labeled trees on n nodes and designed an $O(n^6)$ time algorithm based on dynamic programming. Shasha and Zhang [24] presented a faster $O(n^4)$ time algorithm for the same problem. Then, Klein [18] considered the more general problem of computing the edit distance between unrooted edge-labeled trees and further improved the complexity to $O(n^3 \log n)$. The improved algorithm and the solution given by Shasha and Zhang are both based on a recursive formula, which reduces computing the edit distance between two trees to computing the edit distance between two smaller trees. This needs to be done carefully as to restrict the number of different trees that will appear during the whole process, and the gist of Klein's improvement is a nice application of the heavy path decomposition

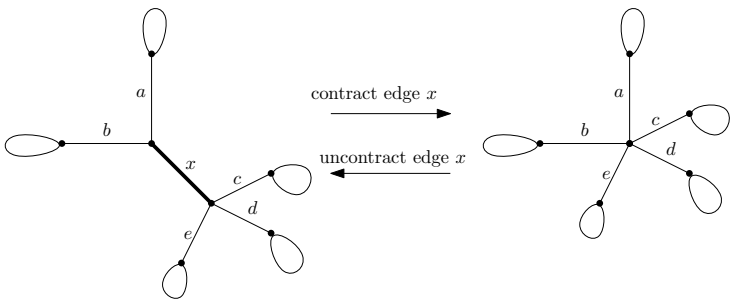

Figure 1: Contraction and uncontraction of the edge with label x costs $c_{del}(x) = c_{ins}(x)$.

to do so. 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 [13]. Finally, Demaine et al. [12] further improved the complexity for rooted node-labeled trees to $O(n^3)$ and showed that this is optimal among all decomposition strategies. Their idea, again at a high level, was to apply the heavy path decomposition to both trees. 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. However, they were not able to generalize their algorithm to unrooted trees.

Although Demaine et al. [12] showed that their algorithm is optimal among all decomposition strategies, it is not clear that any algorithm for computing the edit distance 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 [20] present an experimental comparison of the known algorithms. Aratsu et al. [3], Akutsu et al. [1], and Ivkin [16] design 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. [8], who showed that significantly improving the cubic time complexity for rooted node-labeled trees is not possible under some popular conjectures.

While computing the edit distance between rooted trees seems well understood by now, the best solution for unrooted trees is still $O(n^3 \log n)$. While, by a simple reduction (see Section 8), unrooted trees are at least as difficult as rooted trees, we do not know if they are strictly more difficult. However, a straightforward modification of the algorithm by Demaine et al. [12] to unrooted trees results in a much higher time complexity. Even the seemingly easier instance, in which one of the trees is a full binary tree, and the other is a caterpillar already cannot be solved by Demaine et al.'s approach efficiently. See Figure 2 for illustration of this case.

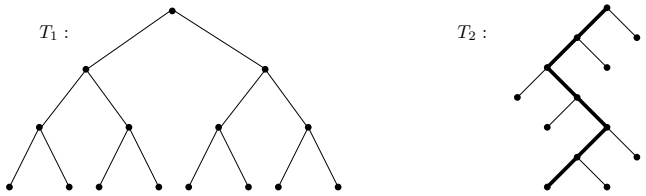

Figure 2: Full binary tree and a caterpillar.

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 $m \leqslant n$, it runs in $O(nm^2(1 + \log \frac{n}{m}))$ time and $O(nm)$ space. It matches the complexity of Demaine et al.'s algorithm for the rooted case and improves Klein's algorithm for the unrooted case. As all the rooted lower bounds also apply to the unrooted case, our algorithm is optimal among all decomposition algorithms, and it is unlikely that there exists a significantly faster approach, unless some popular conjecture fails. Additionally, we provide a new description and analysis of the Demaine et al.'s algorithm.

Our approach for the unrooted case is based on both Klein's and Demaine et al.'s algorithms but requires some additional techniques. We start with heavy path decomposition of one of the trees, and for each of its heavy paths, we process the other tree differently. It reminisces the approach of [7, 11], in which some nodes are more important than the others. On top of that, we use divide and conquer approach with an additional telescoping trick.

Roadmap. In Section 2 we introduce the recursive formula and the naming convention we use. In Section 3 we present some of the known algorithms for the rooted case. Next, in Section 4 we

return to the unrooted case, introduce new notation and transform both the input trees adding some auxiliary edges.

Then, in Section 5 we present our new $O(n^3 \log\log n)$ algorithm for the unrooted case which already improves the Klein's algorithm and is essential for understanding our main $O(n^3)$ algorithm described in Section 6. Both of the new algorithms are first described for the case when one of the trees is a single heavy path and then generalized. Later, in Section 7 we focus on implementation details, how to preprocess the trees and limit the space to $O(nm)$. Finally, in Section 8 we apply all the known lower bounds for the rooted case to the unrooted one.

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/uncontracting an edge labeled with α. For every $\alpha, \beta \in \Sigma$, we know the cost $c_{match}(\alpha, \beta) = c_{match}(\beta, \alpha)$ 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 elementary operations transforming T_1 to T_2, where an elementary operation is contracting/uncontracting an edge or changing a label of an edge.

There are two versions of the tree edit distance problem (TED). One is the rooted case, in which both trees are rooted, and every node has its children ordered left-to-right. The second is unrooted, in which each node has its neighbors cyclically ordered, and we can think that the trees are embedded in the plane, as they are unrooted.

Note that the edit distance between unrooted trees is a minimum over edit distance between all possible rootings of T_1 and T_2, where a rooting is uniquely determined by choice of the root of the tree and the leftmost edge from the root. Thus there are $2(n-1)$ possible rootings of an unrooted tree with n nodes. Since costs of contraction and uncontraction of an edge are equal, we can only consider the case in which only contractions and relabelings are allowed. Then we transform both T_1 and T_2 to a common tree, and from an optimal sequence of operations in this problem, we can easily obtain an optimal sequence of operations transforming T_1 into T_2 and vice versa.

We first suppose, that both trees are of equal size $n = |T_1| = |T_2|$, but in the end, we will also address the problem, when one of the trees is significantly larger than the other. In the beginning, we focus only on the case of rooted trees which is essential for understanding our algorithm for the unrooted case.

There is also another version of TED problem, in which labels are on nodes instead of edges. In the rooted case, the operation of deletion of a (non-root) node is defined as merging it with its parent. Those versions are easily reducible to each other in the rooted case, however, in the unrooted case, the result of deletion of a node is ambiguous, as it depends on a rooting, which is missing. Thus, later on, we can focus only on the case with labels on edges.

2.1 Naming convention

Recall, that we start with the case when both trees are rooted. We use a similar naming convention as in [12]. We call main left and right edges of a 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 similarly except that we consider the left main edge of T. T^v denotes subtree of T rooted in node v. See Figure 3.

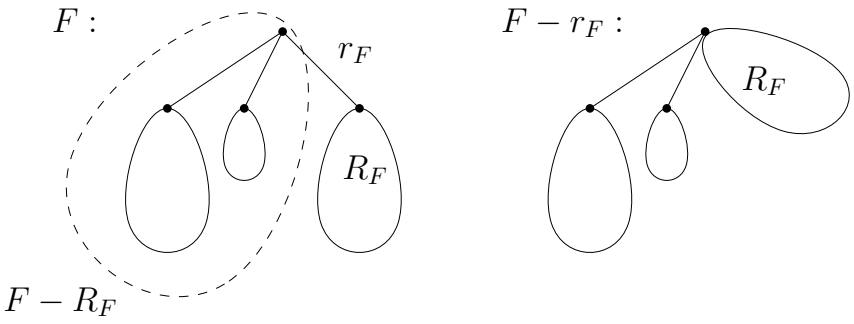

Figure 3: Tree F with its right main edge contracted (right) and both r_F and R_G contracted (left).

Given a rooted tree T, by E_T we denote the Euler tour of T, that is a sequence of edges obtained during DFS traversal of T started from the root of T and visiting nodes in left-to-right order. Every edge of T appears exactly twice in E_T. One occurrence corresponds to traversing the edge down the tree and the other to traversing it up the tree.

We define a pruned subtree of a tree T to be a tree obtained from T by a sequence of contractions of the left or right main edge of T. Next, an interval is any contiguous subsequence of E_T. Consider an interval W and observe, that after removing from W all edges that appear exactly once in it, it becomes exactly $E_{T'}$ for some T', a pruned subtree of T. Conversely, for every pruned subtree T' of T, there is at least one interval $W_{T'}$ of E_T, which is $E_{T'}$ with possibly some other edges appearing exactly once in it. Thus, later on, we will identify every pruned subtree T' of T with the shortest interval on E_T corresponding to T', which is a unique representation.

Observe that for a pruned subtree T', the first edge appearing twice in $E_{T'}$ is the left main edge of T', and the last one is the right main one. Consequently, we can also think that a non-empty pruned subtree is represented by its left and right main edge.

To conclude, there are two equivalent representations of a pruned subtree T' of T: either as an interval on E_T or as its left and right main edges. In both cases, we can completely represent a pruned subtree in $O(1)$ space. 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.

2.2 Dynamic programming

Shasha and Zhang [24] introduced the following dynamic programming approach for computing the edit distance between two rooted trees:

Lemma 2.1. *Let $\delta(F, G)$ be the edit distance between two pruned subtrees F and G of respectively T_1 and T_2. Then:*

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

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

It tries contracting the right main edge in only one of the two trees or matching 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 4 for an illustration of this case.

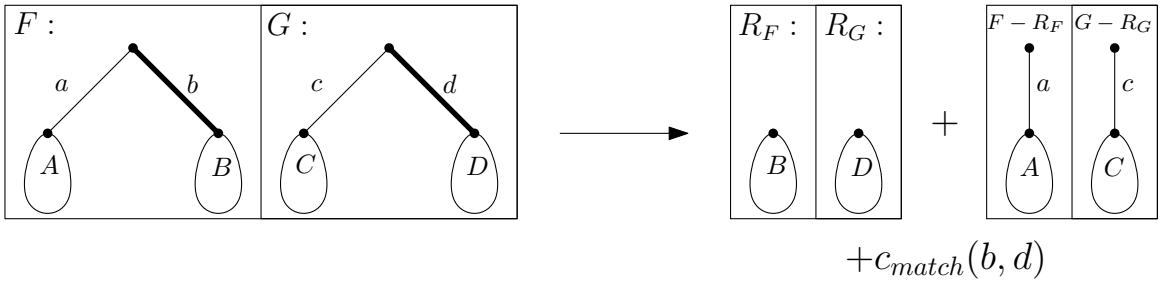

Figure 4: In the case when both right main edges are not contracted, we obtain two independent problems.

To estimate time complexity of the algorithm, we only count different pairs (F, G) for which $\delta(F, G)$ is computed because the formula from Lemma 2.1 can be evaluated in constant time. Some values of $\delta(F, G)$ might be retrieved multiple times, but using memoization we can assure that every $\delta(F, G)$ is computed at most once. How to memoize it efficiently is a secondary concern and will be addressed separately in Section 7.1.

Note that in every call of $\delta(F, G)$, F is a pruned subtree of T_1, thus it corresponds to an interval on E_{T_1}. As $|E_{T_1}| = O(n)$, there are $O(n^2)$ different possible pruned subtrees of T_1. Similarly arguing for T_2 we conclude that there are at most $O(n^4)$ different subproblems visited by Shasha and Zhang's algorithm.

3 Edit distance between rooted trees

In the above algorithm, in every recursive call, we always contract or relabel the right main edge. However, 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 subproblems considered. Such a family of algorithms is called decomposition algorithms:

Definition 3.1. ([12]) A decomposition algorithm for computing edit distance between rooted trees T_1 and T_2 is an algorithm based on the recurrence from Lemma 2.1 using an arbitrary strategy.

The first improvement of Shasha and Zhang's algorithm was made by Klein, with $O(n^3 \log n)$ algorithm [18]. This algorithm is also capable of solving the unrooted case in the same time, which is currently the fastest known approach for unrooted TED. Demaine et al. [12] presented algorithm running in $O(n^3)$ time for equal-sized rooted trees and $O(nm^2(1 + \log \frac{n}{m}))$ for trees of size n, m where $m \leqslant n$ and proved that it is optimal among all decomposition algorithms.

Based on the two approaches we develop an intermediate, $O(n^3 \log \log n)$ algorithm, which is not optimal but will be useful for understanding our algorithm for the unrooted case.

3.1 Klein's $O(n^3 \log n)$ algorithm

Suppose that F and G are pruned subtrees of respectively T_1 and T_2. Klein [18] uses the following strategy to compute $\delta(F, G)$: if $|L_F| < |R_F|$ then choose left direction, otherwise right. Note that the choice does not depend on G.

To upper bound the number of subproblems visited by Klein's algorithm, we bound the number of visited pruned subtrees (call them *relevant intervals*) of T_1 and T_2 separately. Then the total number of visited subproblems will be bounded by product of the two numbers. Clearly, as argued for Shasha and Zhang's algorithm, there are $O(n^2)$ relevant intervals of T_2.

To upper bound the number of relevant intervals of T_1 we use heavy path decomposition [25] of T_1. In this technique, we call the root of the tree light and every node chooses its child with the largest subtree (and the leftmost in case of ties) and calls it heavy and all other children light. We say that an edge is heavy if it leads to the heavy child. Note that heavy edges induce a decomposition of a tree into paths, every heavy path starts in a light node and ends in a leaf of the tree and all nodes apart from the top one on a heavy path are heavy. See Figure 5.

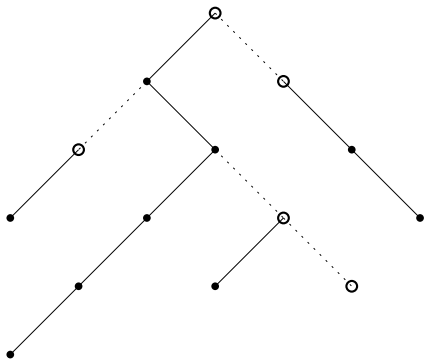

Figure 5: Decomposition of a tree into heavy paths. Heavy edges are marked with solid lines and heavy nodes with full circles. Light edges are dotted and light nodes are empty.

Observe that contracting or relabeling 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. We define that strategy "avoiding the heavy child" chooses the direction in such a way that the edge leading to the heavy child of the root is contracted or relabeled as late as possible. If there is a choice, when the strategy can choose both the left and right main edge (as none of them leads to the heavy child), then we define that the strategy chooses the left one. Similarly, when there is no choice, and there is only one child from the root, as for now, we define that strategy chooses the left direction. This strategy will be used in all our algorithms.

Now we show that using the strategy in T_1, no relevant interval can be obtained only by a recursive call of the form $F - L_F$ or $F - R_F$, as each of them is also obtained by a sequence of contractions of an edge according to the strategy.

Lemma 3.2. *Consider an arbitrary tree T. Suppose that strategy avoiding the heavy child in T says L for a pruned subtree F. Then $F - L_F$ is also obtained by a sequence of contractions of the main edge according to the strategy.*

Proof. There are two cases to consider. First, if there is only one edge outgoing from the root of F, then left and right main edges overlap and $F - L_F = \varnothing$, so clearly the lemma holds. Otherwise, if there are at least two main edges in F, then contracting the edge not leading to the heavy child of F does not change the heavy child in F. Thus, the strategy will repeatedly say L, until all the edges from L_F are contracted, so finally $F - L_F$ is also obtained, and the lemma holds. □

The lemma implies that to count the relevant intervals of T_1 we can only consider the trees obtained by contraction of the left or right main edge (according to the strategy avoiding the heavy child) and trees of the form L_F and R_F.

Observation 3.3. Consider an arbitrary tree T and one of its heavy paths H with top node u. Then for every node v on H, T^v is obtained from T^u by a sequence of contractions of a main edge according to strategy avoiding the heavy child.

We denote $\mathsf{apex}(F)$ as 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. Intuitively we can say that $\mathsf{apex}(F)$ is the top node of the heavy path containing the root of F, but we need the definition with edges because it is unclear, which node is the result of contraction of an edge. In the following lemma we show that grouping by apex-es the pruned subtrees visited while applying the strategy leads to $\sum_{v\text{: light node in } T_1} |T_1^v|$ bound on their total number:

Lemma 3.4. *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.*

Proof. Consider a visited pruned subtree F with $\mathsf{apex}(F) = v$, where v is a light node. From Lemma 3.2 and Observation 3.3 we conclude, that F is obtained by a (possibly empty) sequence of contractions of a main edge according to the strategy. As there are $|T^v|$ contractions of a main edge until the tree becomes empty, the lemma holds. □

Finally, we need to bound the sum of sizes of subtrees rooted in light nodes of T_1. For that purpose, we denote light-depth $\mathsf{ldepth}(u)$ of a node u as the number of light nodes that are ancestors of u (node is also an ancestor of itself). Note that size of subtree rooted in node u is at most $n/2^{\mathsf{ldepth}(u)-1}$, because if a node v is light, then $2 \cdot |T^v| < |T^w|$, where w is the parent of v. Thus $\mathsf{ldepth}(u) \leqslant \log(n) + 1$ and:

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

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, so runs in $O(n^3 \log n)$ time.

3.2 Intermediate $O(n^3 \log\log n)$ algorithm

In this section, we present an algorithm running in $O(n^3 \log\log n)$ time for the rooted case. It is not optimal (as Demaine et al.'s run in $O(n^3)$), but will be useful to understand our $O(n^3 \log\log n)$ algorithm for the unrooted case. We slightly modify Klein's algorithm to first avoid the heavy child in the pruned subtree of T_1 and when the subtree is small enough, then we switch to avoiding the heavy child in the pruned subtree of T_2. More precisely, we fix a value b and during the computation of $\delta(F, G)$ use the strategy avoiding the heavy child either of F or G, where the choice depends on F and b. Let r be the root of F and H be the heavy path in T_1 containing r. Now we test size of the subtree rooted in the top node h of H. If the subtree rooted in h contains more than n/b nodes, then avoid the heavy child of F, otherwise of G. See Figure 6. In other words, we test size of the subtree rooted in the lowest light ancestor of r. As every node is also its own ancestor, we have $\mathsf{apex}(h) = h$ for every light node h in T_1. Using the apex notation we have that $\mathsf{apex}(F) = \mathsf{apex}(T_1^r) = h$. Observe that we avoid the heavy child of F if F is big enough or F was obtained by a sequence of contractions from a "big" T_1^h according to strategy avoiding the heavy child. Notice that we allow switching strategy to the other subtree only in the last case of the dynamic program in Lemma 2.1, in the recursive call of $\delta(R_F, R_G)$ or $\delta(L_F, L_G)$, because it is the only moment when $\mathsf{apex}(F)$ can change.

To bound the number of subproblems visited, we separately consider the case when the strategy avoids heavy child of F and G. Note that when the strategy avoids heavy child of G,

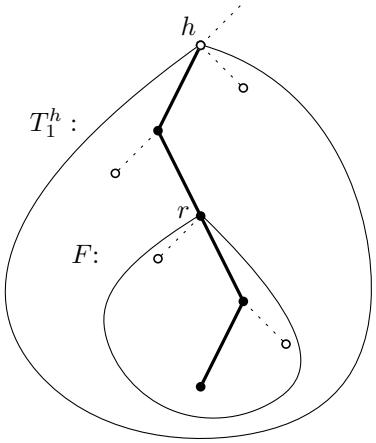

Figure 6: Let empty circles denote light nodes end full ones heavy and $h = \mathsf{apex}(F)$. Then the strategy avoids the heavy child of F if $|T_1^h| > n/b$.

then the size of F is not larger than n/b. Consequently, there are at most $O(n^2/b)$ relevant intervals of T_1, because they correspond to intervals of length at most n/b on Euler tour E_{T_1}, which is of length $O(n)$. From the analysis of Klein's algorithm, there are at most $O(n \log n)$ visited relevant intervals of T_2, so in total there are $O(n^3 \frac{\log n}{b})$ subproblems visited while avoiding the heavy child of G.

While avoiding the heavy child of F, all pruned subtrees of T_2 can be visited, so there can be $O(n^2)$ of them. However, now we visit fewer subtrees of T_1 than in Klein's algorithm. Again, visited pruned subtrees are grouped by their apexes, but these nodes must be of ldepth smaller than $\log b + 1$ to have its subtree greater than n/b. Consequently, in that case, there are $O(n^3 \log b)$ subproblems visited.

To conclude, the algorithm visits $O(n^3 \log b + n^3 \frac{\log n}{b})$ subproblems, which is minimized for $b = \log n$ and results in overall complexity $O(n^3 \log \log n)$.

3.3 Demaine et al.'s $O(n^3)$ algorithm

A natural improvement of Klein's algorithm is to alternate between trees T_1 and T_2 while avoiding the heavy child. However, the main difficulty is to simultaneously alternate between the trees and control the number of visited subproblems. On a high level, Demaine et al. [12] avoid the heavy child in currently larger of the two considered pruned subtrees, however once decided to avoid the heavy child in on of the trees (say T_1), then they avoid the heavy child in this tree until it becomes empty. As in the $O(n^3 \log \log n)$ algorithm, they allow switching strategy to the other subtree is in the last case of the dynamic program in Lemma 2.1, in the recursive call of $\delta(R_F, R_G)$ or $\delta(L_F, L_G)$.

Now we describe in detail the algorithm of Demaine et al. Our presentation is different from the original, is based on the above $O(n^3 \log \log n)$ approach and adapted to the case of TED with labels on edges instead of nodes. Again, the algorithm uses the dynamic program from Lemma 2.1 and to compute $\delta(F, G)$ needs to choose a direction, either left or right, for further computations. The strategy is avoiding the heavy child of X, where X is either F or G. However, now the condition is slightly more involved: if either of F or G is empty, then X is the non-empty of the two, otherwise if $|T_1^{\mathsf{apex}(F)}| > |T_2^{\mathsf{apex}(G)}|$ then X is F, else G. Recall that $\mathsf{apex}(F)$ is the top node of the heavy path containing the lowest common ancestor of all edges in F, as in Figure 6.

Algorithm 1 Our presentation of Demaine et al.'s algorithm [12] for rooted TED.

function $\delta(F, G)$
  if $F = G = \varnothing$ **then return** 0
  if $G = \varnothing$ **or** $(F \neq \varnothing$ **and** $|T_1^{\mathsf{apex}(F)}| > |T_2^{\mathsf{apex}(G)}|)$ **then**
    $X := F$
  else
    $X := G$
  if right child of the root of X is not the heavy one **then**
    return $\min \begin{cases} \delta(F - r_F, G) + c_{del}(r_F) & \text{if } F \neq \varnothing \\ \delta(F, G - r_G) + c_{del}(r_G) & \text{if } G \neq \varnothing \\ \delta(R_F, R_G) + \delta(F - R_F, G - R_G) + c_{match}(r_F, r_G) & \text{if } F, G \neq \varnothing \end{cases}$
  else
    return $\min \begin{cases} \delta(F - l_F, G) + c_{del}(l_F) & \text{if } F \neq \varnothing \\ \delta(F, G - l_G) + c_{del}(l_G) & \text{if } G \neq \varnothing \\ \delta(L_F, L_G) + \delta(F - L_F, G - L_G) + c_{match}(l_F, l_G) & \text{if } F, G \neq \varnothing \end{cases}$

Now we assume that both trees are of size n, $X = F$ and analyze the complexity of the strategy avoiding the heavy child in X. Clearly, $|F|, |G| \leqslant |T_1^{\mathsf{apex}(X)}|$. Observe that F was obtained from $T_1^{\mathsf{apex}(F)}$ by a sequence of successive contractions according to the strategy. Next, $F - R_F$ is the tree F with the right main edge contracted many times. Thus, both $F - r_F$ and $F - R_F$ are also obtained from $T_1^{\mathsf{apex}(F)}$ by a sequence of successive contractions according to the strategy and $\mathsf{apex}(F - r_F) = \mathsf{apex}(F - R_F) = \mathsf{apex}(F)$. Then, the only subsequent recursive call $\delta(F', G')$ in which $\mathsf{apex}(F') \neq \mathsf{apex}(F)$ is due to the recursive call $\delta(R_F, R_G)$ or $\delta(L_F, L_G)$. It may also hold that $\mathsf{apex}(R_F) = \mathsf{apex}(F)$ when F has only one child, and the strategy chooses the edge leading to the heavy child of the root.

We say, that a subproblem $\delta(F, G)$ is charged to the node $\mathsf{apex}(X)$. Consider a node v in T_1. Suppose that v is light because otherwise nothing is charged to it. If v is heavy, then there is no subproblem charged to v, so now suppose that v is light. From all the above observations we have, that among all subproblems (F, G) charged to v, there are $O(|T_1^v|)$ different pruned subtrees F of T_1. Next, all pruned subtrees G of T_2 are not bigger than $|T_1^v|$ and each of them corresponds to an interval of Euler tour of length n. Thus, there are $O(n|T_1^v|)$ different pruned subtrees of T_2 among subproblems charged to node v and in total there are $O(n|T_1^v|^2)$ subproblems charged to a light node v of T_1.

Now summing it over all light nodes of T_1, we obtain that there are $O(n \sum_{v\text{: light node in } T_1} |T_1^v|^2)$ subproblems visited when $X = F$. Because a symmetric argument holds for $X = G$, it remains to upper bound $t(n) := \sum_{v\text{: light node in } T} |T^v|^2$ where T is a tree of size n. Denoting by n_i the total size of the i-th subtree connected to the heavy path containing the root of T, we obtain the following bound: $t(n) \leqslant n^2 + \sum t(n_i)$. It holds that $n_i \leqslant (n-1)/2$ as the i-th subtree is rooted in a light node and $\sum_i n_i \leqslant n - 1$ as the subtrees connected to the heavy path are disjoint. Now we prove by induction that $t(n) \leqslant 2n^2$.

Using the inequality: $a^2 + b^2 \leqslant (a-1)^2 + (b+1)^2$ for $a \leqslant b$ we can upper bound the sum $\sum_i n_i^2$ with $2 \cdot (n/2)^2$ iteratively choosing distinct indices i, j such that $0 < n_i \leqslant n_j < (n-1)/2$, decreasing n_i and increasing n_j by 1. Combining it with the recurrence relation and the induction hypothesis we get:

$$t(n) = \sum_{v\text{: light node in } T} |T^v|^2 = n^2 + \sum_i t(n_i) \leqslant n^2 + 2 \cdot \sum_i n_i^2 \leqslant n^2 + 2 \cdot 2 \cdot (n/2)^2 = 2n^2$$

We conclude that there are $nt(n) = O(n^3)$ subproblems visited when $X = F$ and similarly for

$X = G$. The algorithm can be also proved to run in $O(nm^2(1+\log\frac{n}{m}))$ time for trees of unequal sizes $m \leqslant n$, separately considering light nodes (apexes) u such that $|T^u| \leqslant m$ and $|T^u| > m$.

3.4 Bottom-up perspective

In this section, we rephrase all the above algorithms so that the computation is done in the bottom-up order. The aim of all the algorithms is to compute $\delta(T_1, T_2)$ knowing only $\delta(\varnothing, \cdot)$ and $\delta(\cdot, \varnothing)$, as the costs of contraction of an arbitrary tree are memorized. For that purpose they compute and memoize $\delta(T_1^u, T_2^v)$ for all nodes $u \in T_1$ and $v \in T_2$ in a table Δ, where $\Delta[u, v] := \delta(T_1^u, T_2^v)$.

The main subroutine of these algorithms is to compute $\delta(T_1^x, \cdot)$ from $\delta(T_1^y, \cdot)$ where x is the parent of y and $\cdot$ denotes all pruned subtrees of T_2. In this subroutine, we iteratively "uncontract" an edge using the recursive formula from Lemma 2.1. Formally, given a pruned subtree F and $\delta(F - r_F, \cdot)$ we would like to compute $\delta(F, \cdot)$. However, to use the recursive formula for computing $\delta(F, G)$ for a pruned subtree G of T_2, we also need the values of $\delta(R_F, R_G)$ and $\delta(F - R_F, G - R_G)$. Let $T_1^{x(i)}$ be the tree obtained from T_1^x after i contractions of a main edge avoiding the heavy child. See Algorithm 2 for the case when y is either the leftmost or rightmost child of x. In the other cases, we need to first uncontract the edges "to the left" of y and then "to the right" of y.

Algorithm 2 Obtains $\delta(T_1^x, \cdot)$ from $\delta(T_1^y, \cdot)$, x is y's parent and $\cdot$ are all pruned subtrees of T_2.

1: **function** ComputeFrom($\delta(T_1^x, \cdot), \delta(T_1^y, \cdot)$)
2: **if** y is the leftmost child of x **then**
3: let k satisfy $T_1^{x(k)} = T_1^y$
4: **for** $i = k - 1 \dots 0$ **do**
5: **for each** pruned subtree G of T_2 **do** ▷ in the order of increasing sizes of G
6:
$$\delta(T_1^{x(i)}, G) = \min \begin{cases} \delta(T_1^{x(i+1)}, G) + c_{del}(r_{T_1^{x(i)}}) \\ \delta(T_1^{x(i)}, G - r_G) + c_{del}(r_G) \\ \delta(R_{T_1^{x(i)}}, R_G) + \delta(T_1^{x(i)} - R_{T_1^{x(i)}}, G - R_G) + c_{match}(r_{T_1^{x(i)}}, r_G) \end{cases}$$
7: **else**
8: analogously for left contractions

Observe, that processing the pruned subtrees in this order we have that $T_1^{x(i)} - R_{T_1^{x(i)}} \in \{\varnothing, T_1^{x(j)}\}$ for some value $j > i$, so we have already computed $\delta(T_1^{x(j)}, \cdot)$. Finally, the algorithms ensure that $\delta(R_{T_1^{x(i)}}, R_G)$ is already computed by processing heavy paths in an appropriate order. Note that there exist nodes a, b such that $\Delta[a, b] = \delta(R_{T_1^{x(i)}}, R_G)$.

Klein's algorithm. Recall that Klein's algorithm avoids the heavy child in T_1 and possibly visits every pruned subtree of T_2. Consider a heavy path H in T_1 with top node u. From Lemma 3.2 we know that every subproblem $\delta(F, G)$ obtained during computing $\Delta[w, v]$ for $w \in H$ and $v \in T_2$ is either of the two types:

(i) $\delta(T_1^x, T_2^y)$ for a light node x attached to H and $y \in T_2$, or

(ii) F is one of the pruned subtrees obtained by a sequence of contractions according to the strategy avoiding the heavy child from T_1^u.

To avoid recursive calls we need to ensure that all the mentioned subproblems are already computed and memorized. First, we process subproblems of the second type (ii) in the order of increasing size of F and memoize all the intermediate results. In other words, if v_i are the nodes on H, then for $i = (|H| - 1)..1$ we run ComputeFrom$(\delta(T_1^{v_i}, \cdot), \delta(T_1^{v_{i+1}}, \cdot))$. We know $\delta(T_1^{v_{|H|}}, \cdot)$, because the tree $T_1^{v_{|H|}}$ is empty, so it is the cost of contraction of all the edges in the pruned subtree of T_2.

Finally, we need an order of processing heavy paths of T_1 to make sure that subproblems of type (i) have been already processed and stored in Δ. Note that it is enough to consider them in the order of decreasing light-depths of their top nodes or in other words, of increasing sizes of subtrees rooted in top nodes of them.

To conclude, by processing the heavy paths in such an order we make sure that there is no need to use recursive calls, provided that we memoize Δ and the intermediate results for every heavy path.

Intermediate $O(n^3 \log\log n)$ algorithm. In this algorithm we first compute $\Delta[u, v]$ for all u satisfying $|T_1^{\mathsf{apex}(u)}| \leqslant n/b$ and $v \in T_2$. In this phase, we can process heavy paths of T_2 in the order as above, but now considering only subtrees of T_1 of size not exceeding n/b.

Then, we use the values computed in the first phase to fill all the remaining fields of Δ. We only have to process the heavy paths of T_1 of light-depth smaller than $\log b$ and use the same order as earlier.

Demaine et al.'s algorithm. The algorithm of Demaine et al. also processes heavy paths in the order of decreasing light-depths, but now we consider heavy paths both from T_1 and T_2 simultaneously. Suppose it considers a heavy path H of T_1 with top node u. Then it computes $\Delta[w, v]$ for all nodes $w \in H$ and $v \in T_2$ such that $|T_2^v| \leqslant |T_1^u|$.

4 Back to unrooted case

Recall that edit distance between two unrooted trees T_1 and T_2 is a 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. We start by showing that not all pairs of possible rootings need to be considered. Klein [18] mentioned, that it is intuitively evident and has been formally proved by Srikanta Tirthapura in personal communication.

Lemma 4.1. *For every rooting of T_1 there is at least one rooting of T_2 such that edit distance between these two rooted trees admits minimum edit distance among all possible rootings of T_1 and T_2.*

Proof. Let $T^\star$ denote the common tree, to which both trees are transformed into in the optimal setting and observe that nodes in $T^\star$ correspond to subtrees of T_1 and T_2 merged to a single node. See Figure 7. Consider an arbitrary rooting of T_1 with r being the root and e being the left main edge from r in T_1 and let $r^\star$ be the node in $T^\star$ to which r is mapped.

If $T^\star$ is empty, then every rooting of T_2 admits the minimum edit distance. Otherwise, observe that there is a natural, induced by T_1 left-to-right ordering of edges incident to $r^\star$ in $T^\star$. Then let $e^\star$ denote the leftmost edge from $r^\star$ in that ordering. Note that not necessarily it is the edge e from T_1, which is mapped to $e^\star$ because e might be contracted in the optimal setting. Next, there is an edge e' from T_2 which is mapped to $e^\star$. Finally, observe that the rooting of T_2 with e' being the leftmost edge from the root together with the considered rooting of T_1 admit the minimum edit distance among all possible rootings of T_1 and T_2. □

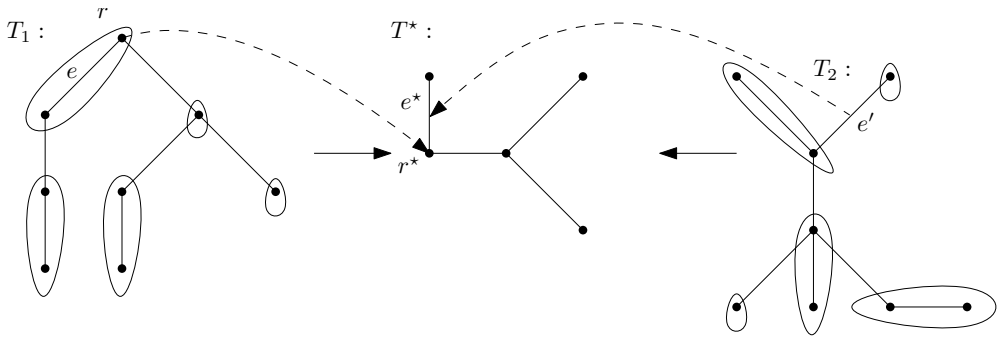

Figure 7: The common tree $T^\star$, to which both trees T_1 and T_2 are transformed, consists of subtrees merged to single nodes. Node r is mapped to $r^\star$ in $T^\star$, edge $e^\star$ is the leftmost from $r^\star$ with respect to the ordering induced by T_1 and e' is the edge from T_2 that is mapped to $e^\star$.

The lemma guarantees that 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. Thus every algorithm for the rooted case can be naively used $O(n)$ times to find the edit distance between unrooted trees. Both Klein's and our approach start with an arbitrary rooting of T_1 and then efficiently try all possible rootings of T_2.

Consider an arbitrary rooting of T_2 and an Euler tour E_{T_2} on it. As E_{T_2} is an Euler tour, then its every cyclic shift is also an Euler tour, but starting from a different edge. Moreover, every rooting of T_2 corresponds to a cyclic shift of E_{T_2}. See Figure 8. Thus we can use Euler tour E_{T_2} of an arbitrary rooting of T_2 and treat it as a cyclic string. Then every possible pruned subtree (in any rooting) of T_2 is an interval of E_{T_2} and there are still $O(n^2)$ possible intervals.

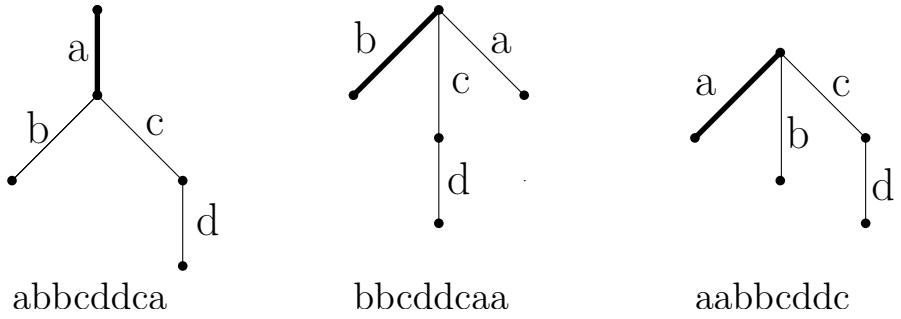

Figure 8: Euler tours of different routings are intervals of any Euler tour treated as a cyclic string, for example: *abbcddca*|*abbcddca*. The left main edge that determines the rooting is the thick one.

All the above observations show, that Klein's algorithm that chooses the direction based only on T_1 can check all possible rootings of T_2 in $O(n^3 \log n)$ time, the same as for the rooted case. It is currently the fastest algorithm for the unrooted case.

4.1 Slight modifications

Apart from treating Euler tours E_{T_1} and E_{T_2} as cyclic strings, we need to introduce new definitions to handle the unrooted case. Recall, that even in the unrooted case, we first arbitrarily root both trees and the initial rooting remains unchanged throughout the algorithm, and all further definitions are according to this rooting.

Darts. For a fixed rooting of a tree, there are two possible directions of traversing an edge e: either from the root ($e^{\downarrow}$) or towards the root ($e^{\uparrow}$). To distinguish between these cases, later on we think that instead of one undirected edge $e = \{u, v\}$ there are two directed edges (darts): $e^{\downarrow} = (u, v)$ and $e^{\uparrow} = (v, u)$ (or oppositely, if v is closer to the root than u). We define subtree $\mathsf{subtree}(d)$ of a dart $d = (u, v)$ to be the subtree rooted in node v of the initial tree, when u is the parent of v. See Figure 9. To locate $\mathsf{subtree}(e^{\downarrow})$ as an interval of Euler tour, observe that it starts right after $e^{\downarrow}$ and ends right before $e^{\uparrow}$ on the cyclic Euler tour and also can be empty.

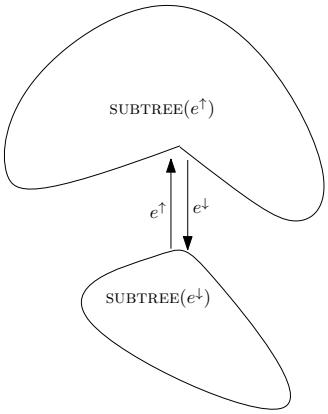

Figure 9: Note that $e^{\uparrow}$ and $e^{\downarrow}$ belong neither to $\mathsf{subtree}(e^{\uparrow})$ nor to $\mathsf{subtree}(e^{\downarrow})$.

We still can use the notion of pruned subtrees – no matter how the tree is rooted, every subtree corresponds to an interval of E_{T_2} and contracting the left or the right main edge results in changing one of the ends of the interval. Thus, every pruned subtree also corresponds to an interval of E_{T_2}, so – even in the unrooted case – there are $O(n^2)$ of them. Similarly, still every pruned subtree can be represented by its left and right main edges. If there are more than one edge from the root, then the representation is unique, see Figure 10. However, if there is only one edge from the root, then one also needs to specify "direction" of the tree whether it is "up" or "down" the edge, see Figure 11. Equivalently, in this situation it is enough to provide a dart instead of edge with direction.

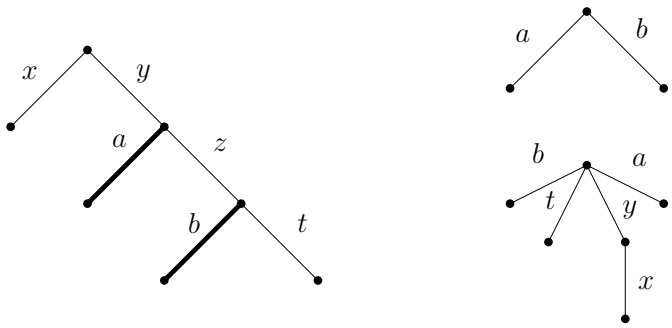

Figure 10: If a pruned subtree has more than one edge from the root, then it is uniquely represented by its left and right main edges.

Auxiliary edges for rootings. As our main goal is to compute the edit distance between two unrooted trees, we will compute $\delta(T_1, T)$ for all T being different rootings of T_2. We observed that every rooting of T_2 corresponds to a subrange of any 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

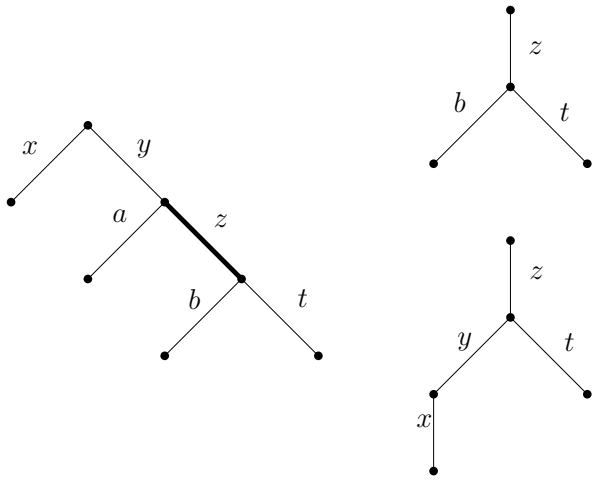

Figure 11: If a pruned subtree has exactly one edge from the root, then one also needs to specify "direction" of the tree.

labeled with a fresh label $\# \notin \Sigma$ which will be used only to denote a rooting. For every node v we add new edges alternating with the original ones. See Figure 12. Thus in total, there are $2(n-1)$ edges added, exactly the number of possible rootings we check. Note that for every new edge, there are two darts and subtree of one of them corresponds to the original tree rooted in a particular way and subtree of the other dart is empty.

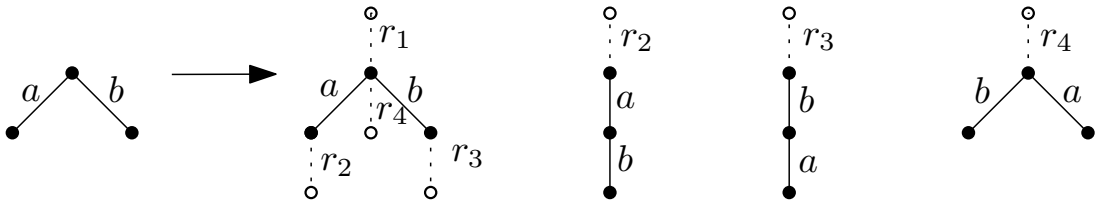

Figure 12: We added 4 edges: $r_1, .., r_4$ to the tree consisting of edges a and b. Each rooting of the original tree corresponds to a dart from one of the new edges.

Observe that in the tree T_1 we effectively use only one rooting, so we can add only one new edge. In order to allow only contraction of those new edges, we set $c_{del}(\#) = 0$ and $c_{match}(\#, \alpha) = \infty$ for all $\alpha \in \Sigma \cup \{\#\}$. Note that addition of the new edges does not change the cost of the optimal solution, as all of them will be contracted with cost 0. Thus in the original dynamic program of Lemma 2.1 we can treat the new edges exactly in the same way as the old ones.

Finally, having an optimal sequence of operations for the modified trees, we should simply discard all contractions of the new edges to obtain the optimal solution for transforming one of the original trees to the other. Observe, that later on, we can forget about labels on edges because they have no impact on the performance of any strategy used by the algorithm. Labels matter only for the exact value of $\delta(\cdot, \cdot)$, but our aim is to minimize the number of pairs (F, G), for which $\delta(F, G)$ is computed.

Using the auxiliary edges for rootings we say that our algorithm for the edit distance between unrooted trees needs to compute $\delta(T_1, \mathsf{subtree}(r_2))$ [1] for all darts r_2 representing rootings of T_2. For that purpose it will compute value of $\delta(\mathsf{subtree}(T_1^u), \mathsf{subtree}(d_2))$ for all nodes $u \in T_1$ and

[1] To simplify notation, we write T_1 instead of $\mathsf{subtree}(r_1)$, where r_1 is the dart corresponding to the initial rooting of T_1.

all darts d_2 in T_2, both up and down T_2. Again all these values will be stored in the table Δ, but now the second index is a dart in T_2: $\Delta[u, d] := \delta(T_1^u, \mathsf{subtree}(d))$. Finally, when the whole table Δ is filled, it is possible to extract the edit distance between (unrooted) T_1 and T_2.

Auxiliary edges to make trees binary. Later it will be useful to have the trees binary, which means that every node has at most two children. To achieve it, we add another set of edges with label #, which costs 0 to contract and ∞ to match with another edge. We need to split nodes with more than two children in such a way, that the order of children is preserved after contraction of the new edges. There are various ways of obtaining it, and one of them is presented in Figure 13. Clearly, there are $O(n)$ nodes and edges added, and after contraction of the new edges (which costs 0), we obtain the original tree.

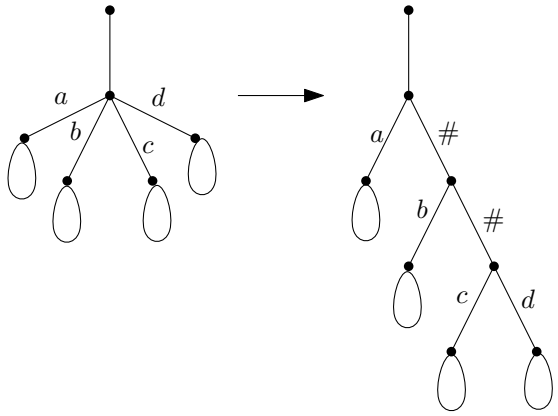

Figure 13: A possible way of making the tree binary from an arbitrary tree with a linear number of additional edges.

All differences between rooted and unrooted case. To sum up, in comparison with the rooted case, now we consider darts instead of undirected edges and treat Euler tour as a cyclic string. We transformed the original trees introducing new edges that will be used to describe rootings of the original tree as a subtree of a new dart and finally made the tree binary. In total, we added $O(n)$ new nodes and edges. Later on by T_1 and T_2 we denote the transformed trees and their size by n. The aim of the following algorithms for the edit distance between unrooted trees is to compute $\delta(T_1, \mathsf{subtree}(r_2))$ for all darts r_2 representing rootings of T_2.

The cost of the optimal solution for the modified trees is the same as for the original ones and having a sequence of operations for the modified trees, we can easily obtain an optimal sequence for the original problem.

5 $O(n^3 \log\log n)$ 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 for the rooted case from Section 3.3 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 Δ. Now we need to fill the remaining fields $\Delta[u, d]$ for all darts d 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 one, smaller subproblem to obtain fewer subproblems to consider in the next

step. We use divide and conquer paradigm, in which there is more and more sharing after every phase.

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. 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. Similarly, as in the $O(n^3 \log\log n)$ algorithm for the rooted case, 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.
- Otherwise apply a new strategy based only on G and T_2.

Considering it bottom-up, the algorithm first fills values of $\Delta[u, d]$ for all nodes u such that $|T_1^{\mathsf{apex}(u)}| \leqslant n/b$ and all darts in T_2. For the remaining fields of Δ, it uses strategy avoiding the heavy child in T_1. As in the bottom-up description of the Klein's algorithm, in this phase, the algorithm needs to process heavy paths of T_1 in the order of decreasing light-depths. Notice that now we can use the analysis of the $O(n^3 \log\log n)$ algorithm for the rooted case from Section 3.2. Using a similar reasoning we conclude that there are again $O(n^3 \log b)$ subproblems visited in total, because even if now T_2 is unrooted, there are still $O(n^2)$ pruned subtrees there.

For the other phase we can also use the observation, that there are $O(n^2/b)$ relevant subtrees in T_1, but 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 $\Delta[u, d]$ for all darts d in T_2 and all nodes $u \in T_1$ such that $|T_1^u| \leqslant n/b$, call them interesting. Clearly, all subproblems in which there is a switch to the strategy based on T_2 are of this form.

Recall that now the second tree T_2 is arbitrarily rooted and nodes are called light or heavy according to this rooting. The notion of going up or down the tree remains unchanged, according to this rooting.

We will describe the algorithm only from the bottom-up perspective to be able to state which subproblems are computed in every step precisely. As the strategy now is more complex than earlier, we first describe it for the case when T_2 is 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 for a single heavy path to handle arbitrary trees T_2.

5.1 Single heavy path T_2

Now we are considering 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, 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 14 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 15.

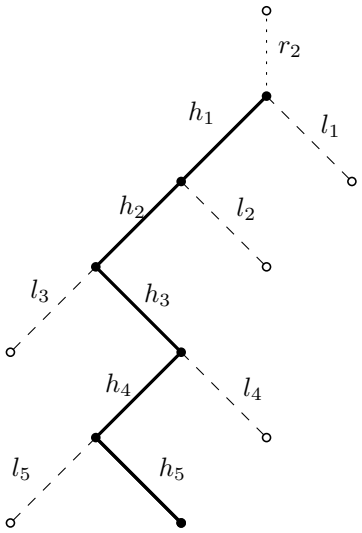

Figure 14: A heavy path H with single connected nodes, where full circles denote heavy nodes and empty circles light nodes. Solid edges are heavy and dotted ones are light. There is also an edge r_2 (dotted) denoting the rooting of T_2.

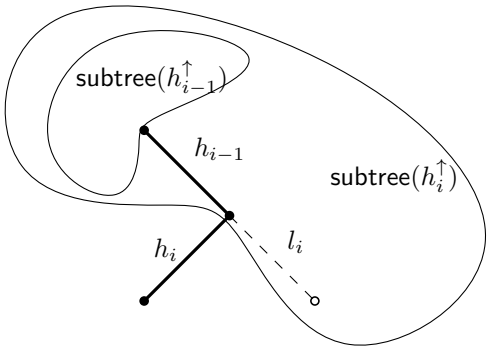

Figure 15: To compute $\delta(*, \mathsf{subtree}(h_i^{\uparrow}))$ we use $\delta(*, \mathsf{subtree}(h_{i-1}^{\uparrow}))$ and uncontract first the edge l_i (if exists).

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 we can retrieve in a constant time), as $h_0 = r_2$ and $\mathsf{subtree}(h_0^{\uparrow}) = \varnothing$. Then, having values of $\delta(*, \mathsf{subtree}(h_{i-1}^{\uparrow}))$ we compute $\delta(*, \mathsf{subtree}(h_i^{\uparrow}))$ by uncontracting appropriate edges. 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 have two edges h_i and h_{i-1} with a common endpoint. Now it is crucial that the strategy in a single step always chooses the same direction. Usually while avoiding the parent or the heavy child there is no choice, but when there is only one edge from the root of the considered pruned subtree, then we define that the strategy chooses the same direction as in other moves in this step. For example, if we compute $\delta(*, T_2^u)$ from $\delta(*, T_2^v)$ where v is the left child of u, then the strategy avoiding the heavy child in T_2 always chooses right direction. Then, for the subtree with only one edge from the root (leading to v), we define that the strategy also chooses right direction. It does not make any difference for pruned subtrees visited from T_2, but from T_1 it does and will matter later.

There are $O(n)$ pruned subtrees of T_2 obtained by uncontractions of the main edge according to the strategy, starting from an empty subtree. Now we need to ensure that the algorithm did

not consider any other pruned subtree of T_2. Suppose it uncontracted the left main edge. Then $G - L_G \in \{\varnothing, G - l_G\}$, depending on whether l_G was the heavy edge leading to the parent or not. Also $L_G \in \{\varnothing, G - l_G\}$, 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 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. As we need these values because they correspond to some rootings of T_2, we will consider them in the following paragraph.

Darts up tree from light nodes. First, we define $\mathsf{merged}_H^R(A, B)$ as the pruned subtree obtained by contraction of edges between the A-th and B-the node on H to the right of H:

Definition 5.1. Let H be a heavy path and A and B ($A \leqslant B$) denote indices of two nodes on H. Then $\mathsf{merged}_H^R(A, B)$ is a tree with the left main edge h_{A-1} and the right main edge h_B. $\mathsf{merged}_H^L(A, B)$ is a tree with the left main edge h_B and the right main edge h_{A-1}.

In other words, $\mathsf{merged}_H^R(A, B)$ is the tree in which there are contracted "all edges between v_A and v_B" on H and to the right of H. See Figure 16. Note that $\mathsf{subtree}(l_A^{\uparrow})$ is either $\mathsf{merged}_H^R(A, A)$ or $\mathsf{merged}_H^L(A, A)$, depending on which side of H is l_i.

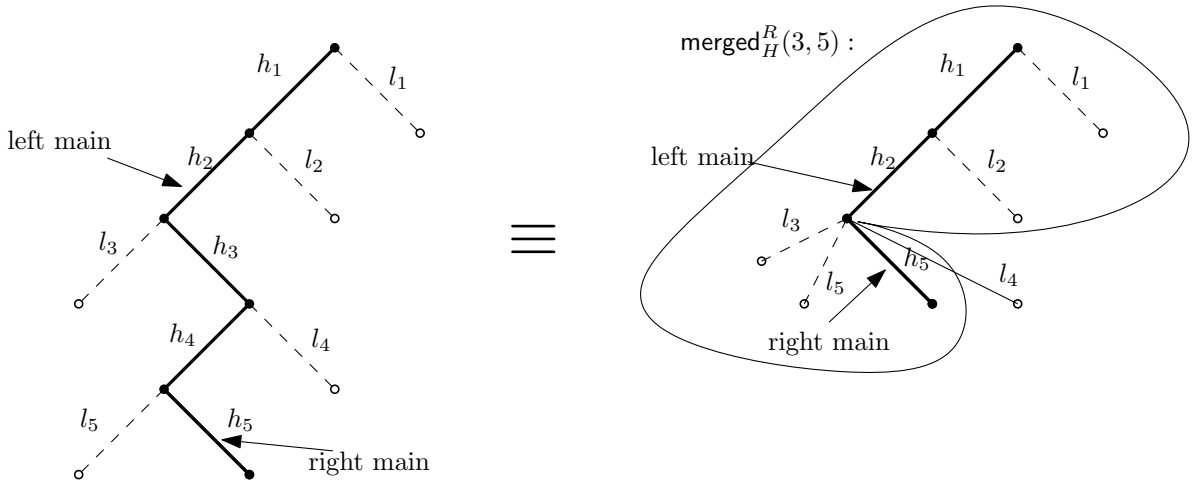

Figure 16: Pruned subtree $\mathsf{merged}_H^R(3, 5)$ has the left main edge h_2 and the right h_5.

On a high level, we start with computing $\delta(*, \mathsf{merged}_H^L(1, |H|))$ and $\delta(*, \mathsf{merged}_H^R(1, |H|))$, then compute $\delta(*, \mathsf{merged}_H^L(1, |H|/2))$ and $\delta(*, \mathsf{merged}_H^R(1, |H|/2))$ and recursively consider the interval $[1, |H|/2]$. Similarly for the interval $[|H|/2+1, |H|]$ and all the subsequently obtained intervals. Eventually we reach interval $[A, A]$ for every value of A and then use $\delta(*, \mathsf{merged}_H^L(A, A))$ or $\delta(*, \mathsf{merged}_H^R(A, A))$ to fill $\Delta[u, l_A^{\uparrow}]$ for all interesting nodes u.

More precisely, we develop a subroutine which, considering an interval $[A, B]$ of indices on H, calls itself recursively for intervals $[A, M]$ and $[M + 1, B]$ where $M = \lfloor \frac{A+B}{2} \rfloor$. As an input it takes "extreme" values of $\delta(*, \mathsf{subtree}(h_{A-1}^{\uparrow})), \delta(*, \mathsf{subtree}(h_B^{\downarrow})), \delta(*, \mathsf{merged}_H^L(A, B))$ and $\delta(*, \mathsf{merged}_H^R(A, B))$, which we later on we denote as $\mathsf{Data}(A, B)$. Then we compute the intermediate ones for $\mathsf{subtree}(h_M^{\downarrow})$,$\mathsf{subtree}(h_M^{\uparrow})$, $\mathsf{merged}_H^L(A, M)$, $\mathsf{merged}_H^R(A, M)$, $\mathsf{merged}_H^L(M + 1, B)$

and $\mathsf{merged}^R_H(M+1, B)$ and call the function recursively. When we reach the case when $A = B$ then can stop and fill some values of Δ. See Algorithm 3.

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

1: **function** $\textsc{Group}_H(A, B, \mathsf{Data}(A, B))$
2: **if** $A = B$ **then**
3: **if** there is a light edge l_A connected to H **then**
4: fill $\Delta[u, l_A^{\uparrow}]$ for interesting nodes $u \in T_1$
5: **return**
6: $M := \lfloor (A + B)/2 \rfloor$
7: **for** $i = (B-1)..M$ **do**
8: $\textsc{ComputeFrom}(\delta(*, \mathsf{subtree}(h_i^{\downarrow})), \delta(*, \mathsf{subtree}(h_{i+1}^{\downarrow})))$ ▷ avoiding the heavy child
9: $\textsc{ComputeFrom}(\delta(*, \mathsf{merged}^R_H(A, M)), \{\delta(*, \mathsf{merged}^R_H(A, B)); \delta(*, \mathsf{subtree}(h_{A-1}^{\uparrow}))\})$
▷ uncontracting the right main edge
10: $\textsc{ComputeFrom}(\delta(*, \mathsf{merged}^L_H(A, M)), \{\delta(*, \mathsf{merged}^L_H(A, B)); \delta(*, \mathsf{subtree}(h_{A-1}^{\uparrow}))\})$
▷ uncontracting the left main edge
11: **call** $\textsc{Group}_H(A, M, \mathsf{Data}(A, M))$

12: similar computations for interval $[M+1, B]$
13: **call** $\textsc{Group}_H(M+1, B, \mathsf{Data}(M+1, B))$

Before we describe in detail the procedure, we show how to compute the initial arguments $\mathsf{Data}(1, |H|)$ for the call of $\textsc{Group}_H(1, |H|, \mathsf{Data}(1, |H|))$. First, $\mathsf{subtree}(h^{\downarrow}_{|H|}) = \mathsf{subtree}(h_0^{\uparrow}) = \varnothing$, so again we use the precomputed cost of contraction of the whole pruned subtree. Then we compute $\delta(*, \mathsf{merged}^L_H(1, |H|))$ and $\delta(*, \mathsf{merged}^R_H(1, |H|))$ from $\delta(*, \mathsf{subtree}(h_0^{\uparrow}))$ by constantly uncontracting respectively the right and left main edge. Thus we have all the input values and can call $\textsc{Group}_H(1, |H|, \mathsf{Data}(1, |H|))$ which will eventually compute $\Delta[u, l_i^{\uparrow}]$ for all light edges l_i to the right of H. See Algorithm 4.

Algorithm 4 Computes input tables needed for processing a heavy path H

1: **function** $\textsc{ProcessHeavyPath}_H(\delta(*, \mathsf{subtree}(h_0^{\uparrow})))$
2: **for** $i = 1..|H|$ **do**
3: $\textsc{ComputeFrom}(\delta(*, \mathsf{subtree}(h_i^{\uparrow})), \delta(*, \mathsf{subtree}(h_{i-1}^{\uparrow})))$ ▷ avoiding the parent
4: fill $\Delta[u, h_i^{\uparrow}]$ for all interesting nodes u
5: $\textsc{ComputeFrom}(\delta(*, \mathsf{merged}^R_H(1, |H|)), \delta(*, \mathsf{subtree}(h_0^{\uparrow})))$ ▷ uncontracting the left main edge
6: $\textsc{ComputeFrom}(\delta(*, \mathsf{merged}^L_H(1, |H|)), \delta(*, \mathsf{subtree}(h_0^{\uparrow})))$ ▷ uncontracting the right main edge
7: **call** $\textsc{Group}_H(1, |H|, \mathsf{Data}(1, |H|))$

Now we need to describe the $\textsc{Group}_H$ procedure more precisely. First, in the loop in line 7 the strategy is to avoid the heavy child, the same as in the rooted algorithms in Section 3.4. As in every step, the procedure considers a constant number of pruned subtrees from T_2, during the whole loop there are $O(M - A)$ visited pruned subtrees of T_2.

The call in line 9 needs more input data than the call in line 7, even though the strategy is always uncontracting the right main edge. Recall that it focuses only on the edges to the right of H. If the dynamic program only tries contracting the right main edge, then it would be possible

to compute $\delta(*, \mathsf{merged}^R_H(A, M))$ only from $\delta(*, \mathsf{merged}^R_H(A, B))$. However, it is not the case when the algorithm matches right main edges of the two trees. Note that for every considered pruned subtree G of T_2, R_G is either $\varnothing$ or $\mathsf{subtree}(r_G^{\downarrow})$. In the first case $G - R_G = G - r_G$, so this pruned subtree is already processed. Although, if $r_G = h_X$ then $R_G = \mathsf{subtree}(r_G^{\downarrow})$, then $G - R_G$ is a pruned subtree, which has not been considered yet. More precisely, its left main edge is h_{A-1} and all edges $h_A, h_{A+1}, h_{A+2}, \ldots, h_{X-1}$ were contracted and all edges l_i for $i \in [A, X]$ to the left of H are connected to the root of the tree. See Figure 17.

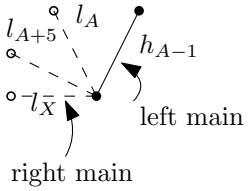

Figure 17: The subtree $G - R_G$ can be of this form, with some remaining edges with indices from $[A, X]$, to the left of the heavy path H.

Note that the largest possible value of X in this step is $X = B - 1$. Observe that all subtrees of this form are obtained by a sequence of uncontractions of the right main edge from $\mathsf{subtree}(h^{\uparrow}_{A-1})$ and that is why we also need $\delta(*, \mathsf{subtree}(h^{\uparrow}_{A-1}))$ in this step and denote it as a set of two input tables. To summarize, in the step in line 9 we need to consider $O(B-A)$ pruned subtrees obtained by uncontraction of right main edge from $\mathsf{merged}_H(A, B)$ and $\mathsf{subtree}(h^{\uparrow}_{A-1})$. All obtained pruned subtrees will be among them. A similar reasoning applies to the edges to the left of H, in line 10. We prove it precisely in Lemma 5.2. Finally, the steps for the interval $[M + 1, B]$ are symmetric.

To sum up, one call of $\textsc{Group}_H(A, B)$ (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.

5.2 Arbitrary tree T_2

Now we need to consider an arbitrary tree T_2, in which there can be arbitrary subtrees connected to the main heavy path. We describe in detail, how to modify Algorithm 4 to process not only a single heavy path, but an arbitrary tree T_2.

In the beginning, the algorithm calls $\textsc{ProcessHeavyPath}_{H^0}(\delta(*, \varnothing))$, where H^0 is the heavy path of T_2 containing the root of T_2. Note that for an arbitrary heavy path H, the procedure only needs to know $\delta(*, \mathsf{subtree}(h^{\uparrow}_0))$ to be able to compute all the input parameters for calls of $\textsc{Group}_H(1, |H|, \mathsf{Data}(1, |H|))$: $\delta(*, \mathsf{subtree}(h^{\uparrow}_0)), \delta(*, \mathsf{subtree}(h^{\downarrow}_{|H|})), \delta(*, \mathsf{merged}^L_H(1, |H|))$ and $\delta(*, \mathsf{merged}^R_H(1, |H|))$. Observe that for every heavy path H $\mathsf{subtree}(h^{\downarrow}_{|H|}) = \varnothing$, so we have $\delta(*, \mathsf{subtree}(h^{\downarrow}_{|H|}))$ precomputed. The only place we need to change inside the $\textsc{Group}_H$ procedure to handle arbitrary trees T_2 is to not only fill $\Delta[u, l^{\uparrow}_A]$ in line 4 of Algorithm 3, but also recursively call $\textsc{ProcessHeavyPath}_{H'}(\delta(*, \mathsf{subtree}(l^{\uparrow}_A)))$ where H' is the heavy path connected to the A-th node of path H. As we pointed earlier, $\mathsf{subtree}(l^{\uparrow}_A)$ is either $\mathsf{merged}^R_H(A, A)$ or $\mathsf{merged}^L_H(A, A)$, depending on which side of H is l_i. Now we need to show that all possibly obtained pruned subtrees are considered:

Lemma 5.2. *In the modified* $\textsc{Group}_H$ *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* $\mathsf{merged}^R_H(A, B)$ *or* $\mathsf{subtree}(h^{\uparrow}_{A-1})$. *A similar property holds for the other three calls of* ComputeFrom *in lines 10 and 12.*

Proof. In line 9 of Algorithm 3 we call $\textsc{ComputeFrom}(\delta(*, \mathsf{merged}^R_H(A, M)), \{\delta(*, \mathsf{merged}^R_H(A, B)); \delta(*, \mathsf{subtree}(h^{\uparrow}_{A-1}))\})$ and suppose the algorithm processes subproblem $\delta(F, G)$. We need to analyze the three subtrees of $T_2 : G - r_G, G - R_G, R_G$ which are obtained using the recursive formula.

Notice that the subtree R_G appears only in the call $\delta(R_F, R_G)$, the value of which we can retrieve from the Δ table. Let $S = G_0, G_1, \ldots, G_k$ be the sequence of subtrees obtained by uncontraction of the right main edge starting from $\mathsf{merged}^R_H(A, M)$, up to $\mathsf{merged}^R_H(A, B)$. Formally, $G_0 = \mathsf{merged}^R_H(A, M)$, $G_k = \mathsf{merged}^R_H(A, B)$ and $G_i = G_{i+1} - r_{G_{i+1}}$. Thus, for every $i > 0$ holds that $G_i - r_{G_i} \in S$. Now we need to consider subtrees $G_i - R_{G_i}$ which also appear in the recursive formula and possibly are not in S.

There are two cases two consider. First, if the right main edge of G_i is not heavy, then $G_i - R_{G_i}$ belongs to S. Otherwise, $G_i - R_{G_i} \notin S$. Let S' be the set of subtrees obtained by a sequence of uncontractions of the right main edge from $\mathsf{subtree}(h^{\uparrow}_{A-1})$. Observe, that in this situation $G_i - R_{G_i}$ belongs to S'. Furthermore, for all subtrees $G' \in S'$ holds that $G' - R'_G \in S'$, which ends the proof for line 9 of Algorithm 3. For computations in the remaining lines, the analysis is symmetric. □

What changes in the analysis of the procedure is that now there are not $O(|H| \log |H|)$ pruned subtrees of T_2 but $O(|T_2^v| \log |H|)$, where v is the top node of H. More precisely, the heavy path H itself might be relatively small, 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 $\textsc{Group}_H$ recursion and thus the bound $O(|T_1^v| \log |H|) = O(|T_1^v| \log n)$.

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

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

5.3 Final analysis

We have just showed an algorithm computing $\Delta[u, e^{\uparrow}]$ for all nodes $u \in T_1$ such that $|T_1^u| \leqslant n/b$ and all darts up the tree T_2, which considers $O(n \log^2 n)$ pruned subtrees of T_2 and $O(n^2/b)$ of T_1. At the beginning of Section 5 we described the second phase of the algorithm, which avoids the heavy child in T_1 and fills the remaining fields of Δ. It considers $O(n \log b)$ pruned subtrees of T_1 and $O(n^2)$ of T_2. Thus, during the two phases, the whole algorithm visited $O(n^3 \frac{\log^2 n}{b} + n^3 \log b)$ subproblems. Now we can choose the value of $b = \log^2 n$ to minimize the above expression, which results in the overall complexity $O(n^3 \log \log n)$. This approach already improves state-of-the-art Klein's $O(n^3 \log n)$ algorithm and will be crucial for the understanding of our $O(n^3)$ algorithm.

5.4 Encoding

Here we describe how to implement this algorithm in $O(n^3 \log \log n)$ space. As we proved, it visits only this number of subproblems, but we need to have a constant-time access to each of them. There are $O(n^4)$ all possible subproblems, and we cannot store their values in one array of

this size. Note that all the computations using the recursive formula from Lemma 2.1 take place in two situations: either while avoiding parent or the heavy child or inside the ComputeFrom procedure. As the reasoning is similar, we describe only the ComputeFrom procedure.

First, the input of the procedure consists of $O(n^2)$ entries, so we can pass them directly. Then, there is a number of subproblems considered, where each of them consists of a pruned subtree of T_1 and T_2. Let C_1 and C_2 be the set of pruned subtrees involved in the computations from respectively T_1 and T_2. At the beginning of the procedure, we can enumerate all elements of the sets C_1 and C_2 and create a local array of size $|C_1| \cdot |C_2|$. Next, for every pruned subtree F from C_1 we should store what is "the index" of $L_F, F - L_F, F - l_F, R_F, F - R_F, F - r_F$ in C_1 (provided it belongs to C_1). Similarly for C_2. Then in the dynamic program, we can look up all the subsequent subproblems in $O(1)$ time.

Recall, that in the analysis of the algorithm we estimated, that there are $O(|C_1| \cdot |C_2|)$ subproblems in this phase, so the bound on memory matches the bound on running time. Thus, the memory used by the algorithm is also $O(n^3 \log\log n)$. In Section 7 we will use a similar technique to show, that it can be implemented even in $O(n^2)$ space.

6 Optimal $O(n^3)$ algorithm for unrooted case

We start with transforming both trees as in the $O(n^3 \log\log n)$ algorithm, that is we add auxiliary edges for rootings, root them arbitrarily and finally make them binary. Let T_1 and T_2 denote the transformed trees. From now on we assume that $|T_1| \leqslant |T_2|$ (otherwise we swap the trees) and let $m = |T_1|, n = |T_2|$. In this section, we present an algorithm that computes the edit distance between unrooted trees in $O(nm^2(1 + \log \frac{n}{m}))$ time.

Again, the algorithm aims to fill the table $\Delta[u, d] := \delta(T_1^u, \mathsf{subtree}(d))$ from which it computes the answer to the original problem. In the beginning we run Demaine et al.'s algorithm [12] which computes $\delta(T_1^u, T_2^v)$ for all pairs of nodes $u \in T_1$ and $v \in T_2$ in $O(nm^2(1 + \log \frac{n}{m}))$ time. Now it remains to compute $\Delta[v, e^{\uparrow}]$ for all nodes $v \in T_1$ and all darts up the tree T_2.

The algorithm first decomposes both trees into heavy paths. Then it processes heavy paths in T_1 in an arbitrary order. The order can be arbitrary, because all the "knowledge" about different paths, that is values of $\delta(T_1^u, T_2^v)$ are already computed by Demaine et al.'s algorithm. To avoid confusion, a heavy path of T_1 we denote by P and of T_2 by H For every heavy path P in T_1 the algorithm fills $\Delta[v, e^{\uparrow}]$ for all nodes $v \in P$ and darts up the tree T_2. Now there is no global parameter b, but instead of that, the algorithm uses m_P, the size of the subtree rooted in the top node of P. See Algorithm 5.

Algorithm 5 Computes the answer in phases.

```
for each P : heavy path in T_1 do
    u := top node of P
    global m_P := |T_1^u|
    fill Δ[T_1^v, e^↑] for all v ∈ P and all darts up the tree T_2
```

We call all the computations for one heavy path a phase. In the following, we describe in detail a single phase. As the presentation is involved, we break it into pieces and gradually handle more and more difficult cases.

Roadmap. In the beginning, similarly as in the $O(n^3 \log\log n)$ algorithm, we first focus on the case when T_2 is a heavy path with single connected nodes, as in Figure 14. It already highlights the difficulties that we will encounter while obtaining $O(nm^2(1 + \log \frac{n}{m}))$ complexity. First, we

also assume that T_1 is a full binary tree, which simplifies the analysis, because there are roughly 2^k heavy paths with size $m/2^k$.

In the next subsection, we relax the assumption on T_1 and consider an arbitrary tree T_1. The change does not affect the algorithm at all (it still runs in phases for every heavy path of T_1), but now we know less about sizes of the heavy paths. In a technical lemma we show that, even in this case, the algorithm also runs in $O(nm^2(1 + \log \frac{n}{m}))$ time.

Next, we adapt the algorithm to handle arbitrary trees T_2, as in the $O(n^3 \log\log n)$ approach. The direct generalization runs in $O(nm^2(1 + \log^2 \frac{n}{m}))$ time, which is already $O(n^3)$, but still slower than Demaine et al.'s algorithm. The next step is to change the way of dividing the interval in the divide and conquer phase and take into consideration, that subtrees connected to one single heavy path have different sizes. This improvement finally leads to $O(nm^2(1+\log \frac{n}{m}))$ running time, which we believe to be optimal.

In the next section we describe how to implement this algorithm in $O(nm)$ space, but as for now, we use $O(n^3)$ space.

6.1 Full binary tree and single heavy path

We first describe the phase for a single heavy path P of T_1 for the case when T_1 is a full binary tree and T_2 is a single heavy path. Recall that u is the top node of P, we defined $m_P = |T_1^u|$ and let H be the single heavy path of T_2.

In the beginning, the algorithm behaves similarly as in the ProcessHeavyPath$_H$ subroutine of $O(n^3 \log\log n)$ approach: it first fills $\Delta[u, h_i^{\uparrow}]$, but now considers all pruned subtrees of T_1^u, which we denote by $\cdot$ in $\delta(\cdot, \mathsf{subtree}(h_i^{\uparrow}))$. The main difference is that it stops the divide and conquer procedure Group$_H$ earlier when the length of the considered interval $B - A$ is smaller than m_P. See Algorithm 6.

Algorithm 6 A slight change in the divide and conquer approach.

```
1: function Group_H(A, B, Data(A, B))
2:     if B − A < m_P then
3:         InsideGroup_H(A, B, ...)
4:         return
5:     M := ⌊(A + B)/2)⌋
6:     compute intermediate values and call recursively as in Algorithm 3
```

At this stage, the algorithm has already computed $\delta(\cdot, \mathsf{subtree}(h^{\uparrow}_{A-1}))$, $\delta(\cdot, \mathsf{subtree}(h^{\downarrow}_{B}))$, $\delta(\cdot, \mathsf{merged}^L_H(A, B))$ and $\delta(\cdot, \mathsf{merged}^R_H(A, B))$ but this is not sufficient to fill all the missing fields of Δ, as $B > A$. To prepare for further computations, the algorithm computes $\delta(\cdot, \mathsf{subtree}(h^{\downarrow}_{B}) \cup \{h_B\})$ from $\delta(\cdot, \mathsf{subtree}(h^{\downarrow}_{B}))$ and $\delta(\cdot, \mathsf{subtree}(h^{\uparrow}_{A-1}) \cup \{h_{A-1}\})$ from $\delta(\cdot, \mathsf{subtree}(h^{\uparrow}_{A-1}))$. Before we describe in detail the InsideGroup$_H$ procedure, we introduce some auxiliary notation.

Auxiliary notation. Let I be the set of edges in T_2 that are "between" h_{A-1} and h_B, formally: $I = \mathsf{subtree}(h^{\downarrow}_{A-1}) \backslash \mathsf{subtree}(h^{\downarrow}_{B}) \backslash \{h_B\}$. See Figure 18 for an example.

Let D be the set of the "distinguished" edges: $D = \{h_{A-1}, h_B\}$ and T_{2D} be a set of four trees with both main edges in D: $T_{2D} = \{G_1, G_2, (\mathsf{subtree}(h^{\uparrow}_{A-1}) \cup \{h_{A-1}\}), (\mathsf{subtree}(h^{\downarrow}_{B}) \cup \{h_B\})\}$ where $l_{G_1} = h_{A-1}, r_{G_1} = h_B$ and $l_{G_2} = h_B, r_{G_2} = h_{A-1}$. Notice that there are in total 6 possible trees with both main edges in D, but we disregard the trees $(\mathsf{subtree}(h^{\uparrow}_{B}) \cup \{h_B\})$ and $(\mathsf{subtree}(h^{\uparrow}_{A-1}) \cup \{h_{A-1}\})$.

Observation 6.1. We have already stored values of $\delta(\cdot, T)$ for all the four trees $T \in T_{2D}$.

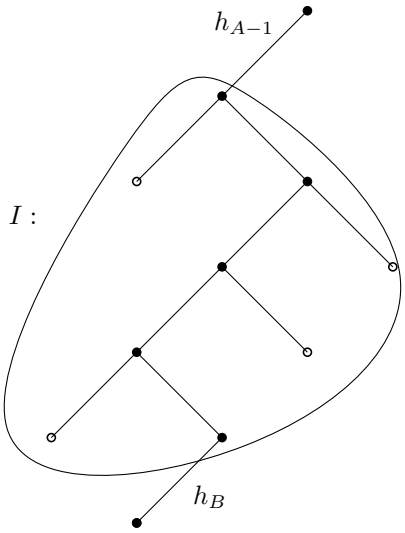

Figure 18: I is the set of edges "between" h_{A-1} and h_B.

Later on, we are interested only in the pruned subtrees with both main edges in $D \cup I$. In some contexts, while specifying a subset of pruned subtrees with specified main edges we will write only a condition on them, for example, $[l_G = h_B]$ means $\{G : l_G = h_B\}$. For instance, the trees from the observation can be written as $[G \in T_{2D}]$, so using this notation, we can rephrase the above observation by saying that we have already computed $\delta(\cdot, [G \in T_{2D}])$. Recall that we use the notation that $T_1^{u(x)}$ denotes the tree T_1^u after x contractions according to the strategy avoiding the heavy child and let $T_1^{u(*)}$ denote all possible pruned subtrees of this form: $T_1^{u(*)} = \{T_1^{u(i)} : i \leqslant |T_1^u|\}$.

Algorithm. On a high-level, the algorithm works in two steps. First, having the values of $\delta(\cdot, [G \in T_{2D}])$, it computes $\delta(\cdot, G)$ for trees G with only one main edge in D always uncontracting an edge from the same side. In the second step, for all the subproblems with both main edges in I, it uses strategy avoiding the heavy child in T_1. In other words it computes then $\delta(T_1^{u(*)}, [l_G, r_G \in I])$. What is crucial, is that when it encounters a tree with at least one main edge in D, then the value of the considered subproblem is already computed, stored and can be returned in a constant time. See Algorithm 7.

Algorithm 7 Fills $\Delta[v, e^{\uparrow}]$ for all edges $e \in I$.

1: **function** $\textsc{InsideGroup}_H(A, B, \delta(\cdot, \mathsf{subtree}(h^{\uparrow}_{A-1}) \cup \{h_{A-1}\}), \delta(\cdot, \mathsf{subtree}(h^{\downarrow}_B) \cup \{h_B\}),$ $\delta(\cdot, \mathsf{merged}^R_H(A, B)), \delta(\cdot, \mathsf{merged}^L_H(A, B)))$
2: $\quad \textsc{ComputeFrom}(\delta(\cdot, [l_G = h_{A-1}]), \{\delta(\cdot, \mathsf{merged}^R_H(A, B)), \delta(\cdot, \mathsf{subtree}(h^{\uparrow}_{A-1}))\})$ ▷ right
3: $\quad\quad$ store $\delta(T_1^{u(*)}, [l_G = h_{A-1}])$
4: $\quad \textsc{ComputeFrom}(\delta(\cdot, [r_G = h_{A-1}]), \{\delta(\cdot, \mathsf{merged}^L_H(A, B)), \delta(\cdot, \mathsf{subtree}(h^{\uparrow}_{A-1}))\})$ ▷ left
5: $\quad\quad$ store $\delta(T_1^{u(*)}, [r_G = h_{A-1}])$
6: $\quad \textsc{ComputeFrom}(\delta(\cdot, [r_G = h_B]), \{\delta(\cdot, \mathsf{merged}^R_H(A, B)), \delta(\cdot, \mathsf{subtree}(h^{\downarrow}_B))\})$ ▷ left
7: $\quad\quad$ store $\delta(T_1^{u(*)}, [r_G = h_B])$
8: $\quad \textsc{ComputeFrom}(\delta(\cdot, [l_G = h_B]), \{\delta(\cdot, \mathsf{merged}^L_H(A, B)), \delta(\cdot, \mathsf{subtree}(h^{\downarrow}_B))\})$ ▷ right
9: $\quad\quad$ store $\delta(T_1^{u(*)}, [l_G = h_B])$
10: $\quad$ **for** $i = (|T_1^u| - 1)..0$ **do**
11: $\quad\quad \textsc{ComputeFrom}(\delta(T_1^{u(i)}, [l_G, r_G \in I]), \delta(T_1^{u(i+1)}, [l_G, r_G \in I]))$ ▷ avoiding the heavy child
12: $\quad\quad$ **if** exists $v : T_1^{u(i)} = T_1^v$ **then**
13: $\quad\quad\quad$ fill $\Delta[v, e^{\uparrow}]$ for all edges $l \in I$

In lines 3, 5, 7 and 9 we highlight, which values will be later used by the algorithm, the other can be discarded. More precisely, in the computations in line 11, if the algorithm considers a tree with one of the main edges in D, then it can use the stored value instead of using the recursive formula.

Observation 6.2. For every tree G with both main edges in I holds that $G - l_G, G - L_G, L_G$ have both main edges in $D \cup I$. A similar property holds for the right direction.

From Observation 6.2 we have that all the computations in line 11 consider only the trees with both main edges in $D \cup I$ and no other kind of tree can appear. So there are $O(m_P^2)$ pruned subtrees of T_2 considered, as $|I| \leqslant 2m_P$ (recall that $B - A < m_P$). Next, while avoiding the heavy child of T_1^u we only consider pruned subtrees of $T_1^{u(*)}$, so there are $O(m_P)$ of them. Thus, during all the computations in the loop in line 10 there are $O(m_P^3)$ considered subproblems. Similarly, in all the earlier computations of the $\textsc{InsideGroup}_H$ procedure, there are $O(m_P^2)$ pruned subtrees of T_1^u and $O(m_P)$ of T_2, so in total there $O(m_P^3)$ subproblems considered in the $\textsc{InsideGroup}_H$ procedure. Notice that for every group size of set I is at least $m_P/2$. The sets are disjoint, so there are at most $2n/m_P$ groups on the heavy path in total. Thus, there are $O(nm_P^2)$ pruned subtrees considered in all calls of the $\textsc{InsideGroup}_H$ procedure.

Now we bound the complexity of the whole $\textsc{ProcessHeavyPath}_H$ procedure, similarly as in the analysis of the $O(n^3 \log\log n)$ algorithm. First, there are $O(1 + \log \frac{n}{m_P})$ recursive calls, because the length of the interval is halved until it gets smaller than m_P. Again, every edge of T_2 contributes to $O(1)$ pruned subtrees on every level of recursion, so there are $O(n(1+\log \frac{n}{m_P}))$ subtrees of T_2. We consider all the $O(m_P^2)$ pruned subtrees of T_1^u, so all the computations during recursive calls visit $O(nm_P^2(1 + \log \frac{n}{m_P}))$ subproblems. Adding $O(nm_P^2)$ pruned subtrees from the calls of the $\textsc{InsideGroup}_H$ procedure we conclude that in total, during the whole phase for one heavy path P of T_1, the algorithm considers $O(nm_P^2(1 + \log \frac{n}{m_P}))$ subproblems. Now we need to sum this over all heavy paths P in T_1. As T_1 is a full binary tree of size m, we can write:

$$\sum_{P\in T_1} nm_P^2\left(1+\log\frac{n}{m_P}\right) \leqslant n\sum_{i=0}^{\log m} 2^i\left(\frac{m}{2^i}\right)^2\left(1+\log\frac{n}{m/2^{i+1}}\right) = nm^2\sum_{i=0}^{\log m}\frac{1}{2^i}\left(2+i+\log\frac{n}{m}\right)$$
$$= nm^2\left(\sum_{i=0}^{\log m}\frac{2+i}{2^i}+\log\frac{n}{m}\sum_{i=0}^{\log m}\frac{1}{2^i}\right) \in O\left(nm^2\left(1+\log\frac{n}{m}\right)\right)$$

To conclude, the algorithm for darts up the tree T_2 visits in total $O(nm^2(1+\log\frac{n}{m}))$ subproblems. Recall that Demaine et al.'s algorithm and the strategy for all darts up T_2 from heavy nodes visit the same number of subproblems. To conclude, we obtain that, in total, the whole algorithm for full binary tree T_1 and a single heavy path T_2 runs in $O(nm^2(1+\log\frac{n}{m}))$ time.

6.2 Arbitrary tree and single heavy path

For the case of an arbitrary tree T_1, the algorithm is the same as above, but now we need a different analysis of the overall running time. For this purpose we first analyze properties of function $f(x) = x^2(1+\ln\frac{n}{x})$ which appears in complexity of various parts of the algorithm.

Lemma 6.3. *Let $f(x) = x^2(1+\ln\frac{n}{x})$. If x, y satisfy $1 \leqslant x \leqslant y < (m-1)/2$, then:*

(i) $f(t)$ is non-decreasing in the range $[1, n/2)$,

(ii) $f(1)+f(x) \leqslant f(x+1)$,

(iii) if $x > 1$ then: $f(x)+f(y) \leqslant f(x-1)+f(y+1)$.

Proof. We prove (i) directly by computing derivative of f:

$$\frac{\partial f(t)}{\partial t} = \frac{\partial(t^2(1+\ln\frac{n}{t}))}{\partial t} = t\left(1+2\ln\frac{n}{t}\right) \geqslant 1+2\ln\frac{n}{n/2} \geqslant 0$$

Similarly, (ii) follows from the definition and inequalities: $\ln(x) \geqslant 1-1/x$ for $x > 0$ and $n \geqslant x+1$:

$$\begin{aligned} f(x+1)-f(x)-f(1) &= (x+1)^2\left(1+\ln\frac{n}{x+1}\right) - x^2\left(1+\ln\frac{n}{x}\right) - 1 - \ln n \\ &= x^2\ln\frac{x}{x+1} + 2x\left(1+\ln\frac{n}{x+1}\right) + 1 + \ln\frac{n}{x+1} - 1 - \ln n \\ &= x^2\ln\frac{x}{x+1} + 2x\left(1+\ln\frac{n}{x+1}\right) + \ln\frac{1}{x+1} \\ &\geqslant x^2\left(1-\frac{x+1}{x}\right) + 2x + (1-(x+1)) = 0 \end{aligned}$$

To prove (iii) we first show it for $x = y$, that is: if $x > 1$ then holds $2f(x) \leqslant f(x-1)+f(x+1)$.

Now we also need that $n \geqslant m > 2x$:

$$\begin{aligned} f(x+1) + f(x-1) - 2f(x) &= x^2 \ln \frac{x^2}{x^2-1} + 2x \ln \frac{x-1}{x+1} + 2 + \ln \frac{n^2}{x^2-1} \\ &\geqslant x^2 \ln \frac{x^2}{x^2-1} + 2x \ln \frac{x-1}{x+1} + 2 + \ln \frac{(2x)^2}{x^2-1} \\ &\geqslant x^2 \left(1 - \frac{x^2-1}{x^2}\right) + 2x \left(1 - \frac{x+1}{x-1}\right) + 2 + \ln 4 \\ &= 1 - \frac{4x}{x-1} + 2 + \ln 4 \\ &= \frac{-4}{x-1} + \ln 4 - 1 \geqslant 0 \quad \text{for } x \geqslant 12 \end{aligned}$$

For $x < 12$ we calculate the exact value of the expression in the second line which is non-negative. As we proved that (iii) holds for $x = y$, now it is enough to show that:

$$\frac{\partial}{\partial y} \left(f(y+1) + f(x-1) - f(x) - f(y)\right) \geqslant 0.$$

This can be done as follows:

$$\begin{aligned} \frac{\partial}{\partial y}\big(f(y+1) + f(x-1) - f(x) - f(y)\big) &= (2y+2) \ln \frac{n}{y+1} - 2y \ln \frac{n}{y} + 1 \\ &= 2y \ln \frac{y}{y+1} + 2 \ln \frac{n}{y+1} + 1 \\ &\geqslant 2y \left(1 - \frac{y+1}{y}\right) + 2 \ln \frac{n}{y+1} + 1 \\ &= 2 \ln \frac{n}{y+1} - 1 \\ &\geqslant 2 \ln 2 - 1 \geqslant 0 \end{aligned}$$

□

Recall that we need to bound the sum $n \sum_{P\text{: heavy path in } T_1} \left(m_P^2 \left(1 + \log \frac{n}{m_P}\right)\right)$, but now we have no assumptions on the tree T_1. For this we will use the following lemma:

Lemma 6.4. *Let m be size of a tree T and n be an arbitrary number such that $n \geqslant m$. Then:*

$$\sum_{P\text{: heavy path in } T} m_P^2 \left(1 + \log \frac{n}{m_P}\right) = O\left(m^2 \left(1 + \log \frac{n}{m}\right)\right)$$

Proof. We start with changing the logarithm to natural to simplify the following calculations, hiding the constant factor under O. Let $t(m) := \sum_{P\text{: heavy path in } T} m_P^2 \left(1 + \ln \frac{n}{m_P}\right)$ be the above sum for a tree T of size m. Denoting by m_i the total size of the i-th subtree hanging off the heavy path containing the root of T, we obtain the following bound:

$$t(m) \leqslant m^2 \left(1 + \ln \frac{n}{m}\right) + \sum_i t(m_i) \quad \text{where } m_i \leqslant \frac{m-1}{2} \text{ and } \sum_i m_i \leqslant m - 1 \tag{2}$$

where the sum is over all subtrees connected to the heavy path from the root of T.

Now we use the Lemma 6.3 to bound from above the sum $\sum_i f(m_i)$, in which $m_i \leqslant (m-1)/2$ and $\sum_i m_i \leqslant m - 1$ (from (2)). As long as there are three non-zero m_is, two of them are less than $(m-1)/2$ and we choose distinct indices i, j such that $1 \leqslant m_i \leqslant m_j < (m-1)/2$. Then, depending whether m_i equals 1 or not, we apply (ii) or (iii) to decrease m_i and increase m_j by one,

not changing sum of the values. Hence, in the end, there are at most two non-zero m_i's, and they are less than or equal to $m/2$. Next we apply (i) and finally obtain that: $\sum_i f(m_i) \leqslant 2f(m/2)$.

Using the above bound, equation (2) and applying the master theorem we obtain that $t(m) \in O(m^2(1 + \ln \frac{n}{m}))$. □

The lemma finishes the analysis of the algorithm for an arbitrary tree and a single heavy path, which runs in $O(nm^2(1 + \log \frac{n}{m}))$ time.

6.3 Both trees arbitrary

Now we consider the case when both trees are arbitrary. First, we generalize the algorithm for a single heavy path to arbitrary trees, as in the $O(n^3 \log \log n)$ approach. We start with the heavy path H^0 containing the root of T_2 and call $\textsc{ProcessHeavyPath}_{H^0}$, but need to change the $\textsc{Group}_H$ subroutine slightly. Now we do not terminate if $B - A$ is small enough because possibly there are significantly more than m_P edges between them. Instead, we break recursion if there is less than m_P edges "between" h_{A-1} and h_B, as earlier and in Figure 18. We denote $\mathsf{between}(A, B)$ as this set of edges. Moreover, when we reach $A = B$ and there is a heavy path H' connected to H in the A-th node, then we recursively call $\textsc{ProcessHeavyPath}_{H'}$. See Algorithm 8.

Algorithm 8 Divide and conquer approach which terminates if the interval is smaller than m_P.

```
1: function GROUP_H(A, B, Data(A, B))
2:     if |between(A, B)| < m_P then
3:         call INSIDEGROUP_H(A, B, ...)
4:         return
5:     if A = B then
6:         call PROCESSHEAVYPATH_{H'}(δ(·, subtree(l_A^↑)))
7:         return
8:     M := ⌊(A + B)/2⌋
9:     compute intermediate values and call recursively as in Algorithm 3
```

Note that in line 6 we have $\delta(\cdot, \mathsf{subtree}(l_A^{\uparrow}))$, because $\mathsf{subtree}(l_A^{\uparrow})$ is either $\mathsf{merged}_H^L(A, A)$ or $\mathsf{merged}_H^R(A, A)$, depending on which side of H is l_i. We need to notice, that during computations in line 3 we do not have computed values $\Delta[u, e^{\uparrow}]$ for edges e on heavy paths connected to H. However, all subproblems of this form have both main edges in $\mathsf{between}(A, B)$, so they are considered by the $\textsc{InsideGroup}_H$ subroutine and the missing fields of Δ are filled then.

Notice that the changes in the procedure for a single heavy path H of T_2 do not affect the complexity, which is still $O(n_H m_P^2(1 + \log \frac{n_H}{m_P}))$ where n_H is the size of the subtree of T_2 rooted in the top node of H. Thus all the computations for a single heavy path P run in time:

$$\sum_{H\text{: heavy path in } T_2 \text{ s.t. } n_H \geqslant m_P} n_H m_P^2 \left(1 + \log \frac{n_H}{m_P}\right) \in O\left(n m_P^2 \left(1 + \log^2 \frac{n}{m_P}\right)\right)$$

because $\sum_H n_H = O(n(1 + \log \frac{n}{m_P}))$ as the algorithm considers only heavy paths H of T_2 such that $n_H \geqslant m_P$. Now using Lemma 6.4 we obtain, that the overall complexity of the algorithm for the general case of both trees arbitrary is:

$$\sum_{P\text{: heavy path in } T_1} n m_P^2 \left(1 + \log^2 \frac{n_H}{m_P}\right) \in O\left(n m^2 \left(1 + \log^2 \frac{n}{m}\right)\right)$$

which is $O(n^3)$, as desired.

To reduce the complexity and obtain $O(nm^2(1+\log \frac{n}{m}))$ time, we need to modify the $\textsc{Group}_H$ procedure slightly. Our approach reminisces the telescoping technique from [7] and [11] in which some nodes are less important than the others. Intuitively, considering an interval $[A, B]$ on a heavy path H, we would like to divide it in such a way, to ensure that the big subtrees connected to H are contracted smaller number of times. For this purpose, we need to look at the tree from a different perspective.

Big nodes. Let big nodes be the nodes of T_2 with subtree containing at least m_P edges. Then the big nodes form a top part of T_2 with some small nodes connected to it. See Figure 19. We also define that a big light node is called special if it has no big light descendant.

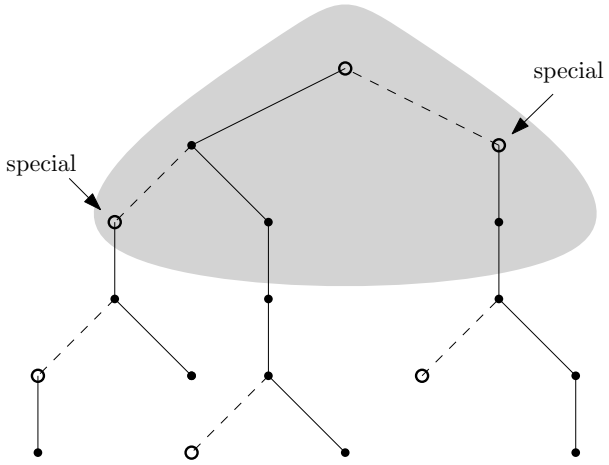

Figure 19: Big nodes constitute the gray upper part of the tree and a big light node is called special if it has no big light descendant.

Now, instead of counting the edges in $\mathsf{between}(A, B)$ we count the special nodes inside subtrees connected to the $A, A+1, \ldots, B$-th node on H and we denote the value as $\mathsf{special}_H(A, B)$. Only when this value is 0, we again focus on $\mathsf{between}(A, B)$. See Figure 20 for an example of how these values are computed.

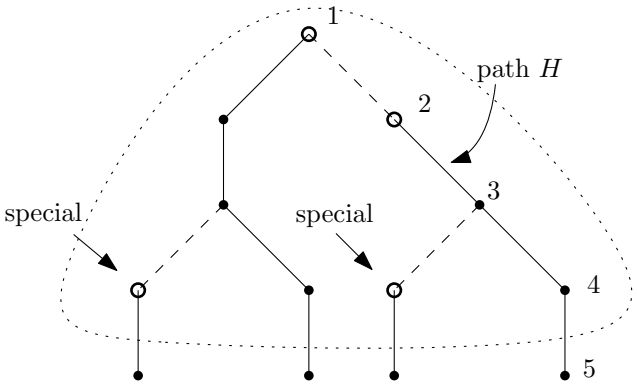

Figure 20: There are two special nodes in the tree and holds: $\mathsf{special}_H(1,3) = 2$, $\mathsf{special}_H(3,5) = 1$ and $\mathsf{special}_H(2,2) = 0$.

Using the notion of special nodes we can describe the algorithm in detail. If for the considered interval $[A, B]$ it holds that $\mathsf{special}(A, B) = 0$, then the pivot is chosen as earlier: $M := \lfloor (A +$

$B)/2)\rfloor$. In the other case we choose M to be the smallest index such that $2 \cdot \mathsf{special}(A, M) \geqslant \mathsf{special}(A, B)$.

Excluding pivot from recursion. Moreover, now we exclude the M-th node from the subsequent recursive calls and run $\textsc{Group}_H(A, M-1)$ and $\textsc{Group}_H(M+1, B)$. The intuition behind it is that the subtree connected to the M-th node will be big (possibly containing much more than m_P edges), so it might be worth not contracting it too often. In addition, we have that $2 \cdot \mathsf{special}(A, M-1) \leqslant \mathsf{special}(A, B)$ and $2 \cdot \mathsf{special}(M+1, B) \leqslant \mathsf{special}(A, B)$, so in the subsequent recursive calls the value of $\mathsf{special}()$ is decreased at least by a factor of two. See Algorithm 9.

Algorithm 9 Divide and conquer approach which terminates if the interval is smaller than m_P.

```
1:  function Group_H(A, B, Data(A, B))
2:      if A > B then
3:          return
4:      if special(A, B) > 0 then
5:          if A = B then
6:              call ProcessHeavyPath_H'(δ(·, subtree(l_A^↑)))
7:              return
8:          else
9:              M := min{k : 2 · special(A, k) > special(A, B)}
10:     else
11:         if B − A < m_P then
12:             call InsideGroup_H(A, B, ...)
13:             return
14:         else
15:             M := ⌊(A + B)/2)⌋
16:     compute arguments for subsequent recursive calls:
17:     call Group_H(A, M − 1, Data(A, M − 1))
18:     call Group_H(M, M, Data(M, M))
19:     call Group_H(M + 1, B, Data(M + 1, B))
```

In the pseudo-code we also call $\textsc{Group}_H(M, M, \mathsf{Data}(M, M))$, but it is only in order to reach one of the final lines 6 or 12. Recall, that all the subroutines $\textsc{ProcessHeavyPath}_H$, $\textsc{InsideGroup}_H$ and the intermediate computations $\textsc{ComputeFrom}$ run in $O(s \cdot m_P^2)$ time (not including subsequent calls), where $s = |I_H(A, B)|$, the number of edges currently considered. So now we need to sum the number of edges considered in all recursive calls together.

Final analysis. Consider an edge $e \in T_2$. Note that all calls $\textsc{InsideGroup}$ consider disjoint sets of edges, so e can contribute to at most one of them. Now we count triples (H', A, B) such that $e \in I_{H'}(A, B)$, that is the edge e is considered in the recursive call $\textsc{Group}_{H'}(A, B)$. Let X_e be the set of such triples. Observe, that there are at most $O(1 + \log \frac{n}{m_P}))$ heavy paths H' on the path from e to the root of T_2 because we consider only heavy paths of size m_P. See Figure 21. Thus we have an upper bound on the number of triples with $A = B$ in X_e.

Now we aggregately count triples $(H', A, B) \in X_e$ such that $A \neq B$. First, observe that there is at most one heavy path H, such that $e \in I_H(A, B)$ and $\mathsf{special}_H(A, B) = 0$ because all the heavy paths "above" have at least one big heavy path connected to them that contains e. In

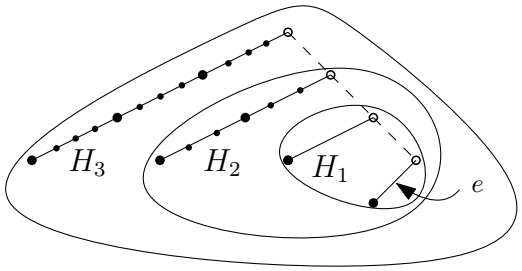

Figure 21: An edge e is considered by all big heavy paths "above" it: H_1, H_2, H_3 and there are $O(1 + \log \frac{n}{m_P})$ of them.

this case, there are at most $O(1 + \log \frac{n}{m_P})$ recursive calls, as every time length of the interval is roughly halved and cannot become smaller than m_P.

Second, as we pointed earlier, every time $\mathsf{special}_H(A, B) \neq 0$, in every subsequent recursive call this value is at least halved. As all the special heavy paths are disjoint and contain at least m_P edges, so there are at most $\frac{n}{m_P}$ of them. Thus, there are also $O(1 + \log \frac{n}{m_P})$ recursive calls with $\mathsf{special}_H(A, B) \neq 0$.

Finally, at the bottom of the recursion we call InsideGroup procedure. All calls are applied on disjoint subsets of edges and each of them consists of at most m_P edges. Recall that if the size of the considered set of edges is x then the complexity of the procedure is $O(m_P^2 \cdot x + m_P \cdot x^2)$. Hence, the total complexity of all these calls is $O(nm_P^2)$.

To conclude, every edge is considered in at most $O(1 + \log \frac{n}{m_P})$ recursive calls, so the total number of edges considered in all recursive calls during the phase for a heavy path P of T_1 is $O(n(1 + \log \frac{n}{m_P}))$. Thus, the whole phase for a heavy path P runs in $O(nm_P^2(1 + \log \frac{n}{m_P}))$ time. Finally we can use Lemma 6.4 to obtain that the whole algorithm computing edit distance between unrooted trees runs in $O(nm^2(1 + \log \frac{n}{m}))$ time.

7 Implementation details

Currently, the above algorithm runs in $O(nm^2(1 + \log \frac{n}{m}))$ time and space. In this section, we show how to implement it in $O(nm)$ space in the same time. Clearly, the tree T_2 is not smaller than T_1, so, later on, we can focus only on it. There will be three difficulties to face.

First, now we cannot preprocess all the pruned subtrees of T_2 because $O(n^2)$ is already too much for us. Thus, given a pruned subtree G of T_2, we need to be able to retrieve in a constant time subtrees $G - l_G, G - L_G, L_G, \ldots$ and the value of $\delta(\varnothing, G)$ (the cost of contraction of all the edges from G). For that purpose, we will use the classic algorithm for Lowest Common Ancestor [5] that runs in linear space and answers queries for the lowest common ancestor (LCA) of two nodes in constant time.

Second, we need to show how to perform the ComputeFrom computations in $O(nm)$ space, which is an order of magnitude less than the number of subproblems considered in the subroutine: $O(nm^2)$. This step will be done similarly as in Demaine et al.'s algorithm, even though now we consider the unrooted case.

Finally, we need to take into the consideration the depth of the recursion of $\textsc{Group}_H$ procedure and count how much data is kept on the stack. We will show, that on every level of recursion there is $O(m^2)$ data stored. As we proved, there are $O(1+\log \frac{n}{m})$ levels of recursion, so using the fact that $\log x \leqslant x$ we get that the total memory kept on the stack is $O(m^2(1+\log \frac{n}{m})) = O(nm)$.

7.1 Preprocessing

Recall that every pruned subtree tree G is represented by its left and right main edges (l_G and r_G). If they overlap, then the tree is of the form $\mathsf{subtree}(d)$ for some dart of T_2. There are only $O(n)$ trees of this form, so we can preprocess them all, that is for every pruned subtree $G = \mathsf{subtree}(d)$ we store $\delta(\varnothing, G)$ and the pruned subtrees: $G - l_G, G - L_G, L_G, \ldots$. Now we focus on one rooting of T_2 but do not have to decompose it into heavy paths. We first run the preprocessing phase that will allow us later to find $LCA(a, b)$ of two arbitrary nodes a, b in a constant time and overall linear space, see [5].

Intermediate subtrees. We first show, how to retrieve pruned subtrees $l_G, G-l_G, L_G, G-L_G$ of a pruned subtree G, for the right side it will be symmetric. Clearly, we already have l_G, because we store the pruned subtree G as its both main edges l_G and r_G. Similarly, L_G is simply $\mathsf{subtree}(l_G^{\uparrow})$ or $\mathsf{subtree}(l_G^{\downarrow})$, depending where is r_G with respect to l_G.

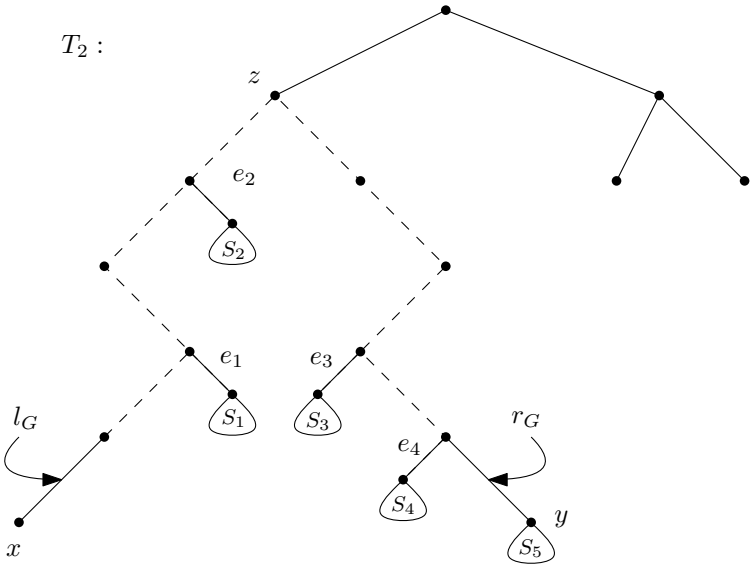

Figure 22: A pruned subtree G with main edges l_G and r_G has all the dashed edges contracted and to the tree belong edges e_i and their subtrees S_i for $i = 1 \ldots 5$.

Let x and y be respectively the first and last nodes on the path from l_G to r_G and $z = LCA(x, y)$ be their lowest common ancestor. See Figure 22. To obtain $G' = G - L_G$ we only need to find its the left main edge, because the right main one does not change (provided that $l_G \neq r_G$). There are two cases: either $l_{G'}$ is connected to right of the path $x..z$ or to the left of the path $z..y$. In the first case it is enough to remember for every node the first edge that is connected to the left and to the right to its path to the root of T_2 and then we can check if the edge is below the node z or not. Otherwise, let t be the leftmost leaf in the right subtree of z (preprocessed, found by traversing down the tree going always left if possible, otherwise right). In Figure 22 the node t is inside subtree S_3. Then $l_{G'}$ is the edge leading to the left child of $LCA(t, y)$. As for the subtree $G - l_G$, if L_G is empty, then $G - l_G = G - L_G$, otherwise we return the left main edge of $\mathsf{subtree}(l_G^{\downarrow})$.

Cost of contraction. Now we show how to retrieve the value of $\delta(\varnothing, G)$, the cost of contraction of all the edges from G. Note that it is the sum of costs of contraction of all edges "to the right" of the path between l_G and r_G plus $c_{del}(l_G) + c_{del}(r_G)$. For example, in Figure 22, we

need to contract l_G and r_G and all edges e_i and their subtrees S_i for $i = 1 \dots 5$. Observe that to contract all edges of G we need to contract all edges to the right of the path $(z \dots x)$ and to the left of $(z \dots y)$. Also notice that the path $(z \dots x)$ is effectively the path $(root \dots x)$ without its prefix $(root \dots z)$. Thus we can use prefix sums and for every node store only the cost of contraction of all edges to the left or right to the path from the root of T_2 to the node.

Note that the above observations hold for all possible pruned subtrees of T_2, for instance also in the case for a subtree G' such that $l_{G'} = r_G$ and $r_{G'} = l_G$. To conclude, it is enough to remember a constant number of values in every node to be able to retrieve all intermediate pruned subtrees and compute the cost of contraction of all edges of a pruned subtree of T_2 in a constant time.

7.2 Computations in limited space

In this subsection, we describe how to implement the ComputeFrom procedure in $O(nm)$ space. Observe that each time we call ComputeFrom subroutine, there is a set of pruned subtrees of one tree (either T_1 or T_2) and two pruned subtrees of the other: the initial and target. For instance, when we call ComputeFrom$(\delta(\cdot, \mathsf{subtree}(h_6^{\uparrow})), \delta(\cdot, \mathsf{subtree}(h_5^{\uparrow})))$, then actually there are considered all pruned subtrees "down" T_1, $\mathsf{subtree}(h_5^{\uparrow})$ is the initial tree and $\mathsf{subtree}(h_6^{\uparrow})$ is target. By a pruned subtree "down" T_1 we denote a pruned subtree obtained by a sequence of contractions of a main edge from the root, starting from T_1 [2]. Clearly in this example there are considered $O(m^2)$ pruned subtrees of T_1 and $O(n)$ of T_2. What is important, is that the target tree is obtained from the initial one by a sequence of uncontractions of a main edge always in the same direction. Later on, we assume, that in this step we always uncontract the left main edge.

As in Section 5.4 we first enumerate all pruned subtrees that are considered during this step to be able to retrieve subsequent trees in the dynamic program in constant time. Now the difficulty lies in the fact that we cannot create the table of size $O(nm^2)$ and we overcome it using an approach based on the one described by Demaine et al. [12]. On a high level, we fix a right main edge of pruned subtree F of T_1 and consider all possible left edges of F. Then there are $O(m)$ candidates for l_F and $O(n)$ candidates for pruned subtree G of T_2. The key insight is that while contracting the left main edge of a tree, its right main edge does not change unless it overlaps with the left one (which is the case when there is only one edge from the root). Using this observation, we can store only the $O(nm)$ values at any time. However, we need to describe the details carefully.

We first describe in detail implementation of the ComputeFrom subroutine for the case when the strategy considers only pruned subtrees "down" T_1, what is sufficient to implement all the algorithms for tree edit distance between rooted trees in $O(nm)$ space. Then we show how to handle also pruned subtrees "up" T_2, which is needed in our algorithms for tree edit distance between unrooted trees.

Pruned subtrees "down" T_1. It might be easier to think, that now we describe how to implement ComputeFrom$(\delta(\cdot, \mathsf{subtree}(h_5^{\downarrow})), \delta(\cdot, \mathsf{subtree}(h_6^{\downarrow}))$ in $O(nm)$ space. In the beginning, we provide an equivalent definition of trees "down" T_1 that will be useful to implement the ComputeFrom subroutine, see Figure 23 and Lemma 7.1.

Lemma 7.1. *For every non-empty pruned subtree F "down" T_1 holds that either $l_F = r_F$ or l_F is strictly to the left of the path from the root to r_F.*

[2] Recall that by T_1 in this context we again denote $\mathsf{subtree}(r_1)$, where r_1 is the dart corresponding to the initial rooting of T_1.

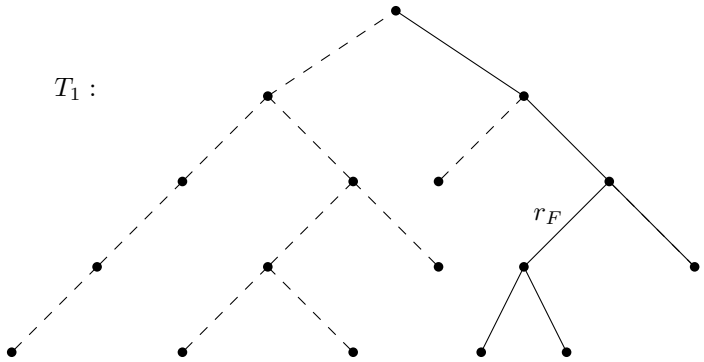

Figure 23: Every pruned subtree F "down" T_1 has its left main edge l_F either equal to r_F or to the left of the path from root to r_F. All the candidates for l_F are marked with the dashed edges.

Proof. It is enough to show that for every pruned subtree F with this property, $F - l_F$ and $F - r_F$ also have this property. This holds from the analysis of three cases: when $l_F = r_F$, $l_F \neq r_F$ and subtree of the contracted edge is non-empty or $l_F \neq r_F$ and subtree of the contracted edge is empty. □

Let S_2 be the set of all intermediate pruned subtrees of T_2 obtained by a sequence of uncontractions of the left main edge from the initial tree to target. Now the algorithm considers candidates for the right main edge of the tree in T_1 in bottom-up order. Then it computes edit distance between all pruned subtrees "down" T_1 with the specific right main edge and all trees from S_2. It also needs to store explicitly values of $\delta(F, G)$ for trees F of the form $\mathsf{subtree}(r^{\downarrow}) \cup \{r\}$ for $r \in T_1$ and $G \in S_2$, because the trees with both main edges overlapping need special attention. See Algorithm 10 for details.

Algorithm 10 Divide and conquer approach which terminates if the interval is smaller than m_P.

function ComputeFrom($\delta(\cdot, \text{target}), \delta(\cdot, \text{initial})$)
  S_2 := set of all intermediate pruned subtrees of T_2 between initial and target tree
  create arrays C and D of size $[m][n]$
  create array $RESULT$ of size $[m^2]$
  for each edge $r \in T_1$ in bottom-up order **do**
    $S_1 := \{F :$ "down" T_1 and $r_F = r\}$
    $F' = \mathsf{subtree}(r^{\downarrow}) \cup \{r\}$
    compute $C[F', G] := \delta(F', G)$ for all $G \in S_2$
    $RESULT[F'] := C[F', \text{target}]$
    create array X of size $[m][n]$
    compute $X[F, G] := \delta(F, G)$ for all $F \in S_1, G \in S_2$
    for each $F \in S_1$ **do**
      $RESULT[F] := X[F][\text{target}]$
    u := the endpoint of r that is closer to the root of T_1
    for each $G \in S_2$ **do**
      $D[T_1^u, G] := X[T_1^u, G]$
  return $RESULT$

Clearly, this subroutine runs in $O(nm)$ space. The arrays C and D are partially filled in every step of the main loop: C stores edit distance between trees of T_1 with one edge from the

root and in D there are trees T_1^u where u is the endpoint of r that is closer to the root. Finally, we need to prove that all the required values during computations in lines 8 and 11 are available in the local arrays that we store. In these lines, we process subtrees in the order of increasing sizes. Recall that we assume that we always uncontract the left main edge.

First consider the step in line 11. While computing $\delta(F, G)$ for $F \in S_1, G \in S_2 \backslash \{\text{initial}\}$ where $l_F \neq r_F$ we need to retrieve value of 4 subproblems and we show that each time it is available in one of the local arrays:

- $\delta(F, G - l_G) = X[F, G - l_G]$, because $G - l_G \in S_2$,
- $\delta(F - l_F, G) = X[F - l_F, G]$, because $F - l_F \in S_1$,
- $\delta(F - L_F, G - L_G) = \delta(F', G - L_G) = C[F', G - L_G]$, because $G - L_G \in S_2$,
- $\delta(L_F, L_G) = \Delta[l_F^{\downarrow}, l_G^{\downarrow}]$ – possibly $L_G \notin S_2$, so we need to use the value from Δ computed in an earlier stage.

Similarly, to compute $\delta(F', G)$ for $G \in S_2 \backslash \{\text{initial}\}$ in line 8 we have the following subproblems to consider:

- $\delta(F', G - l_G) = C[F', G - l_G]$, because $G - l_G \in S_2$,
- $\delta(F' - l_{F'}, G) = \delta(L_{F'}, G) = D[L_{F'}, G]$ which has already been computed, because we consider edges r in bottom-up order,
- $\delta(F - L_{F'}, G - L_G) = \delta(\varnothing, G - L_G)$ which we can retrieve in constant time after the preprocessing described in Section 7.1,
- $\delta(L_{F'}, L_G) = \Delta[l_{F'}^{\downarrow}, l_G^{\downarrow}]$ – as above.

In both variants, in the last case of $\delta(L_F, L_G)$ and $\delta(L_{F'}, L_G)$ we used the values from the Δ table, which were computed by Demaine et al.'s algorithm in the very beginning. However, we can also use the same implementation inside Demaine et al.'s algorithm, but then have to carefully analyze, that indeed the used values have already been computed and stored in the table.

Observe, that the very same implementation works for the case, when there are two input tables, for example $\textsc{ComputeFrom}(\delta(\cdot, \mathsf{merged}_H^R(A, M)), \{\delta(\cdot, \mathsf{merged}_H^R(A, B)); \delta(\cdot, \mathsf{subtree}(h_{A-1}^{\uparrow}))\})$. Note that in this case the set S_2 also contains the trees obtained in the $G - L_G$ move and the subsequent ones. It needs to be slightly larger, because we need to ensure that for every $G \in S_2 \backslash \{\text{initial}\}$ both $G - l_G$ and $G - L_G$ belong to S_2, which is the case as proved in Lemma 5.2. Similarly, note that it does not make any difference when we consider only pruned subtrees of T_1 of size bounded from above, i.e., smaller than n/b (in the case marked with $*$).

Pruned subtrees "down" and "up" T_2. Now we need to slightly modify this approach, because in the InsideGroup subroutine we consider also pruned subtrees "up" T_2: in line 11 of Algorithm 7 we call $\textsc{ComputeFrom}(\delta(T_1^{u(i)}, [l_G, r_G \in I]), \delta(T_1^{u(i+1)}, [l_G, r_G \in I]))$. In this case, we need to consider all possible pruned subtrees of T_2 defined by their two main edges from I. We do it in two steps. First, is symmetric to the Algorithm 10 for all pruned subtrees "down" T_2, that is trees G with l_G to the left of the path from root of T_2 to r_G (marked with dashed lines in Figure 23) and the tree $\mathsf{subtree}(r_G^{\downarrow}) \cup \{r_G\}$ (with $l_G = r_G$). The only difference is that now the roles of T_1 and T_2 are switched.

The second step is for all the remaining pruned subtrees of T_2, with the left main edge not to the left of the path from the root to r_G (marked with solid lines in Figure 23). By the tree with $l_G = r_G$ we mean $\mathsf{subtree}(r_G^{\uparrow}) \cup \{r_G\}$. It is done similarly, but now we need to consider edges r in top-down order and store $D'[F, \mathsf{subtree}(e^{\uparrow})]$, to be able to handle also the case of $G' = \mathsf{subtree}(r^{\uparrow}) \cup \{r\}$. We also have to simultaneously fill the missing values of $\Delta[u, e^{\uparrow}]$ inside the procedure, because these have been already filled only for edges from the heavy path H, not those from the connected small subtrees. To conclude, with these two steps we can implement the ComputeFrom subroutine in $O(nm)$ space.

We also need to elaborate more on the InsideGroup procedure, in which there are considered pruned subtrees G such that $l_G, r_G \in I$, but then the subsequent subtrees might have a main edge inside I, in D. However, we have already computed and values of these subproblems in lines 3,5,7 and 9 as mentioned in Observation 6.2, so can retrieve them in a constant time. To sum up, also the InsideGroup and all computations inside Group_H procedure also fit in the desired $O(nm)$ space.

7.3 Total memory on recursion stack

In the previous subsection we showed how to implement all the intermediate computations inside Group_H, InsideGroup and ComputeFrom in $O(nm)$ space. Clearly, these computations are disjoint, that is every time we run ComputeFrom we can use the one and very same tables C, D of size $O(nm)$ and we only need to store separately inputs and outputs to the procedure.

Recall that during all the computations in T_2 we not always consider the whole tree T_1 and its all edges. All the computations take into consideration the heavy path P from T_1 and only the m_P edges from the subtree T_1^u of its top node u. Then we have the following lemma about the size of tables passed to and from the functions:

Lemma 7.2. *For every call of function* Group_H, InsideGroup *or* ComputeFrom*, size of input and returned tables is* $O(m_P^2)$.

Proof. The lemma clearly holds when we have a table of edit distance between a pruned subtree from T_2 and a set of pruned subtrees of T_1^u because there are $O(m_P^2)$ of them. The only case when we consider many pruned subtrees of T_2, is in the ComputeFrom$(\delta(T_1^{u(i)}, [l_G, r_G \in I]), \delta(T_1^{u(i+1)}, [l_G, r_G \in I]))$ call, but then there are also $O(m_P^2)$ of them, as the set I contains at most m_P elements. □

Thus every recursive call pushes $O(m_P^2)$ values on the stack. From the analysis of Algorithm 9 we know that the depth of the recursion is $O(1 + \log \frac{n}{m_P})$, so for the heavy path P there are in total $O(m_P^2 \cdot (1 + \log \frac{n}{m_P}))$ values stored on the recursion stack. Using the inequality $\log x \leqslant x$, we finally obtain that throughout the whole algorithm, there is $O(nm_P)$ values on the stack. Adding the auxiliary tables of the overall size $O(nm)$ which are shared among ComputeFrom calls, table Δ and all the space used by Demaine et al.'s algorithm, we conclude that the whole algorithm computing edit distance between unrooted trees can be implemented in $O(nm)$ space.

Theorem 7.3. *The algorithm computing edit distance between unrooted trees runs in* $O(nm^2(1+\log \frac{n}{m}))$ *time and* $O(nm)$ *space.*

8 Lower bound

In this section, we restate known lower bounds for the problem of the edit distance between rooted trees (call it rooted TED) and prove that they also hold for unrooted trees. First, Demaine et al. [12] proved the following lower bound for decomposition algorithms:

Theorem 8.1 ([12]). *For every decomposition algorithm and $n \geqslant m$, there exist trees F and G of sizes $\Theta(n)$ and $\Theta(m)$ such that the number of relevant subproblems is $\Omega(m^2 n(1 + \log \frac{n}{m}))$.*

which matches the complexity of the algorithm they provided. Recently, Bringmann et al. [8] proved, that a truly subcubic $O(n^{3-\varepsilon})$ algorithm for rooted TED trees with n nodes is unlikely:

Theorem 8.2 ([8]). *A truly subcubic algorithm for tree edit distance on alphabet size $|\Sigma| = \Omega(n)$ implies a truly subcubic algorithm for APSP. A truly subcubic algorithm for tree edit distance on sufficiently large alphabet size $|\Sigma| = O(1)$ implies an $O(n^{k(1-\varepsilon)})$ algorithm for Max-Weight k-Clique.*

which makes the following conjecture probable:

Conjecture 8.3 ([8]). *For any $\varepsilon > 0$ Tree Edit Distance on two n-node trees cannot be solved in $O(n^{3-\varepsilon})$ time.*

8.1 Unrooted case is also hard

Now we show a reduction from rooted TED to the same problem for unrooted trees (unrooted TED). It increases the number of nodes of a tree and size of the alphabet by a constant number, so the lower bounds from the rooted case will also apply for the unrooted case.

Given an instance $I = (T_1, T_2, \Sigma)$ of rooted TED we want to construct an instance $I' = (T_1', T_2', \Sigma')$ of unrooted TED such that, given an optimal solution of I' it is possible to obtain an optimal solution of I. Clearly, it is not enough to set $I' = I$, because it might be possible to change rooting of one of the trees (say T_2') to obtain a smaller edit distance than between rooted T_1 and T_2. That is actually the point in the problem of unrooted TED.

We need a gadget which ensures, that even if we allow all possible rootings, from every optimal rooting of T_1' and T_2' it is possible to obtain an optimal solution for T_1 and T_2. It is enough to add two edges from the root as shown in Figure 24 and appropriately set costs of contraction and relabeling of the fresh labels $\#_i \notin \Sigma$. They must satisfy that the new edges cannot be contracted and must be matched only with the corresponding edge of the same label in the other tree. More precisely, the costs are set as follows: $c_{del}(\#_i) = \infty$, $c_{match}(\#_i, \#_i) = 0$ and $c_{match}(\#_i, \alpha) = \infty$ for $\alpha \neq \#_i$.

Clearly, in every optimal solution OPT' of I', the new edges with labels $\#_i$ are not contracted and are matched with each other. Observe, that no matter how the trees are rooted in OPT', we can rotate both trees simultaneously in such a way, that $\#_1$ is outgoing from the root as in Figure 24. Informally, we can think of holding the trees by the edge with $\#_1$, with the edge with $\#_2$ underneath and the original tree hanging down in the initial rooting.

To conclude, our reduction adds only 2 new nodes to every tree and 2 new fresh labels to the alphabet and allows retrieving an optimal solution of I from an optimal solution of I'. Thus, all lower bounds from the rooted TED hold also for unrooted TED. Particularly, we proved that every decomposition algorithm for unrooted TED runs in $\Omega(m^2 n(1 + \log \frac{n}{m}))$ time which matches the complexity of our algorithm. Finally, it is unlikely that there exists a truly subcubic $O(n^{3-\varepsilon})$ algorithm for unrooted TED.

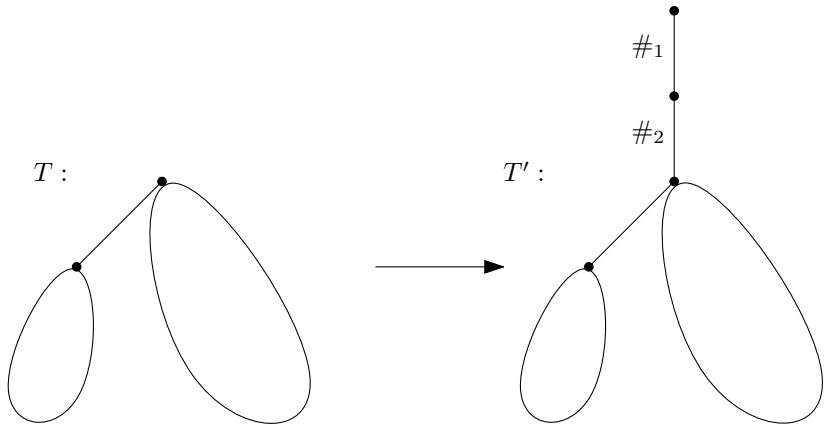

Figure 24: A gadget changing an instance of rooted TED to unrooted TED.

References

[1] T. Akutsu, D. Fukagawa, and A. Takasu. Approximating tree edit distance through string edit distance. *Algorithmica*, 57(2):325–348, Feb. 2010.

[2] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In *IJCAI*, pages 1976–1982, 2013.

[3] T. Aratsu, K. Hirata, and T. Kuboyama. Approximating tree edit distance through string edit distance for binary tree codes. *Fundam. Inf.*, 101(3):157–171, Aug. 2010.

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

[5] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN*, pages 88–94, 2000.

[6] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, June 2005.

[7] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theor. Comput. Sci.*, 11(3):303 – 320, 1980.

[8] K. Bringmann, P. Gawrychowski, S. Mozes, and O. Weimann. Tree edit distance cannot be computed in strongly subcubic time (unless APSP can). *CoRR*, abs/1703.08940, 2017.

[9] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed xml. In *VLDB*, pages 141–152, 2003.

[10] S. S. Chawathe. Comparing hierarchical data in external memory. In *VLDB*, pages 90–101, 1999.

[11] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.

[12] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms*, 6(1):2:1–2:19, Dec. 2009.

[13] S. Dulucq and H. Touzet. Decomposition algorithms for the tree edit distance problem. *J. Discrete Algorithms*, 3(2-4):448–471, 2005.

[14] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33, Nov. 2009.

[15] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, January 1982.

[16] E. Ivkin. Approximating tree edit distance through string edit distance for binary tree codes. B.sc. thesis, Charles University in Prague, 2012.

[17] P. Klein, S. Tirthapura, D. Sharvit, and B. Kimia. A tree-edit-distance algorithm for comparing simple, closed shapes. In *SODA*, pages 696–704, 2000.

[18] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *ESA*, pages 91–102, 1998.

[19] P. N. Klein, T. B. Sebastian, and B. B. Kimia. Shape matching using edit-distance: An implementation. In *SODA*, pages 781–790, 2001.

[20] M. Pawlik and N. Augsten. Efficient computation of the tree edit distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, Mar. 2015.

[21] J. R. Rico-Juan and L. Micó. Comparison of aesa and laesa search algorithms using string and tree-edit-distances. *Pattern Recogn. Lett.*, 24(9-10):1417–1426, June 2003.

[22] T. B. Sebastian, P. N. Klein, and B. B. Kimia. Recognition of shapes by editing their shock graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(5):550–571, May 2004.

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

[24] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms*, 11(4):581–621, Dec. 1990.

[25] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362 – 391, 1983.

[26] K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, July 1979.

[27] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.

[28] X. Yao, B. Van Durme, C. Callison-Burch, and P. Clark. Answer extraction as sequence tagging with tree edit distance. In *NAACL-HLT*, pages 858–867, 2013.