Clemens Andritsch

Tree Edit Distance

Master's Seminar

Graz University of Technology

Institute for Discrete Mathematics
Head: Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Woess Wolfgang

Supervisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Dragoti-Cela Eranda

Graz, July 2018

Contents

1 Introduction

The combinatorial concept of trees appears in many different fields in and outside of mathematics. Especially the problem of comparing two trees comes up in the research of computer languages like structured text databases [10], computer vision and natural language processing. Furthermore, an important application of comparing trees is the analysis of RNA molecules in bioinformatics [7, 1].
This paper is based on the paper *An Optimal Decomposition Algorithm for Tree Edit Distance* by Demaine, Mozes, Rossman and Weimann [3].

1.1 Basic terms and definitions

For the *tree edit distance* problem we are given two rooted labelled ordered trees $F = (V(F), E(F))$ and $G = (V(G), E(G))$. This means, that F and G don't contain any cycles and that siblings have a fixed left-to-right order. Furthermore each node is assigned some *label* from an alphabet Σ. The *tree edit distance* of F and G is the *minimum cost* of transforming F into G. This transformation consists of a sequence of three basic operations: Inserting, deleting or relabelling a node. For a formal definition assume we have a node v with parent v'. The operations do exactly what we intuitively link to their names:
Relabelling changes the label of a node v.
Inserting v connects a new node v to the node v'. The original children of v' get reconnected to become the children of v while keeping the left-to-right order among them.
Deleting is the opposite transformation of inserting. The children of v get reassigned to become the children of v' and v gets deleted. The special case where v is the root of F transforms F into a forest. We denote the forest, which results from deleting the root of F by F°.
An illustration of these operations is given in Figure 1.1. Deleting and inserting are equivalent because deleting a node in F is identical to an

insertion of a node in G. Therefore, we only consider deleting nodes. Since we are talking about some "minimum costs", we have to define some cost functions c_{del} and c_{match}. The cost $c_{del}(\tau)$ defines the cost of deleting or equivalently inserting a node with label τ. $c_{match}(\tau_1, \tau_2)$ describes the cost of changing the label of a node from τ_1 to τ_2.

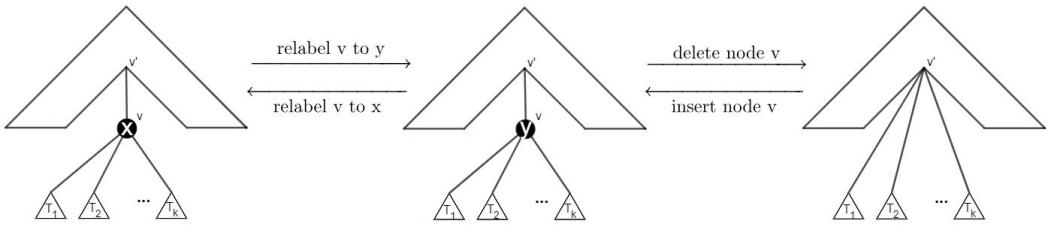


Figure 1.1: The three editing operations on a tree with vertex labels.

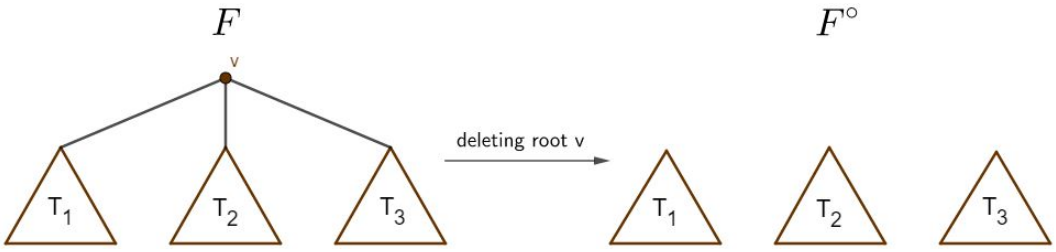
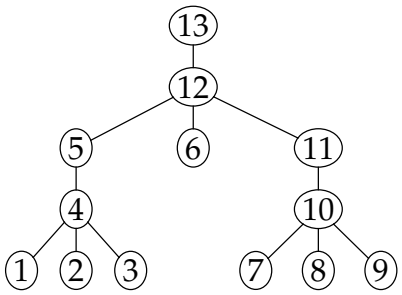


Figure 1.2: Deleting the root v of tree F results in a forest F°.

The previously mentioned order is the so called *post-order indexing*. It is based on the depth first search, but the root of the investigated tree is numbered at last. This is shown in Figure **??**.



Task. For computing the tree edit distance one needs to find a sequence of deletion and relabelling operations on the trees F and G of minimum overall

cost such that after consecutive execution of these operations on F and G both trees become the same. That means they get transformed into two isomorphic forests consistent with the ordering and the labelling of each node. In this paper the tree edit distance of F and G is denoted as $\delta(F, G)$.

As Figure 1.2 suggests, it is possible to delete the root of F. In general this transforms F into a forest denoted by F°. Since this can easily happen during the sequence of transformations, we need to extend the notion of tree edit distance to forests. This extension is trivial by allowing F and G to be some ordered forests. So, for example, $\delta(F^\circ, G^\circ)$ denotes the cost of a minimum cost sequence of operations, such that the forests resulting from deleting the roots in both trees F and G become the same forests. Suppose F to be an ordered forest. Because of the ordering F has a leftmost and a rightmost tree. They are denoted by L_F and R_F respectively and their roots are named l_F and r_F. Furthermore, for some $v \in V(F)$ we define F_v to be the subforest of F rooted at v.

Figure 1.3 shows an illustration of such subtrees with F being the tree in Figure **??**. For easier understanding we name the nodes the same as their post-order index. So the root of F is the node 13, its leaves is the set of nodes $\{1, 2, 3, 7, 8, 9\}$. Furthermore we define $G := F_{12}$ to show the use of R_G and L_G.

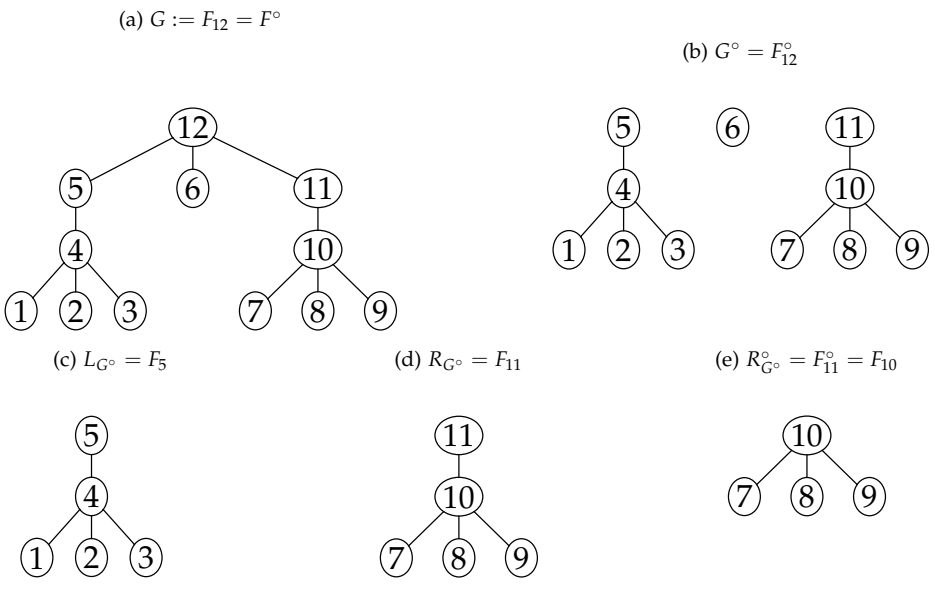


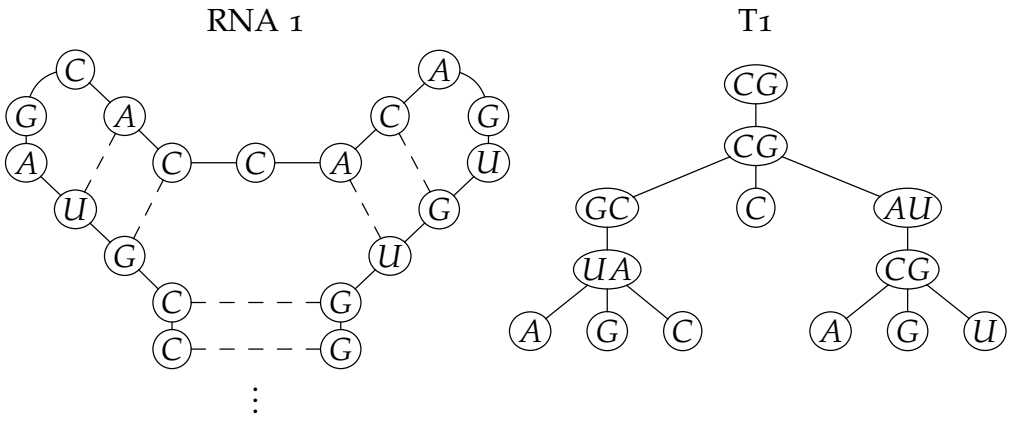
Figure 1.3: Different subtrees using the introduced notion.

Let $n := |V(F)|$, $m := |V(G)|$ be the sizes of the trees such that w.l.o.g. $n \geq m$. Furthermore let n_{leaves} and m_{leaves} denote the number of leaves in the corresponding trees and n_{height} and m_{height} the height of each tree. All those values can be of size $\Theta(n)$ and $\Theta(m)$ respectively.

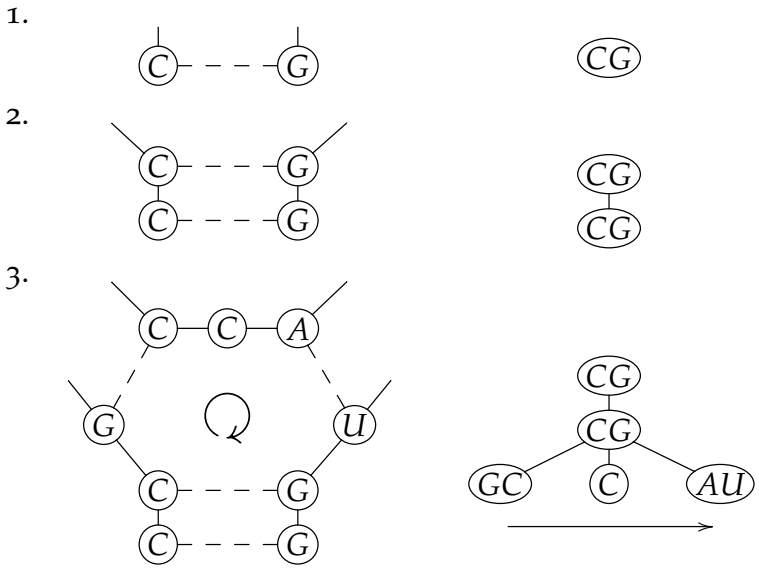
1.2 Small example for RNA structures

As previously mentioned, the comparison of RNA structures is a common application of computing the tree edit distance. Since the 2-dimensional description of an RNA folding is not a tree, we have to find a mapping of such structures to rooted ordered trees. For an easier understanding, we call the RNA folding R and the corresponding rooted labelled tree T. The RNA consists of five different nitrogenous bases called adenine (A), cytosine (C), guanine (G), thymine (T) and uracil (U). In Figures 1.4a and 1.4c we see two examples of such foldings (on the left hand side) and the mapped rooted tree. The dashed lines in those figures represent a hydrogen bond between the two connected nucleotides. These foldings only show some small part of two RNA structures.
Let's discuss the mapping from the RNA folding R to the rooted ordered tree T for the folding in Figure 1.4a. Normally each nucleobase is represented by one node in T, but if two of them are bonded by hydrogen bonds, then their bond is represented by a single node. The folding starts at the bottom with a hydrogen bond (C)–(G), so the root of T has the label (CG). The next nucleotides that we meet is again a hydrogen bond (C)–(G), so we insert a node with label (CG) as a child of the root. If we look at the further structure, we see that we encountered a cycle consisting of the hydrogen bond (G)–(C),a single nucleotide (C) and a hydrogen bond (A)–(U). Therefore we insert one node for each those three elements. The left to right order in the tree T is fixed by the clockwise order of the nucleotides in R. That is why the leftmost node is labelled (GC), the next ons is labelled (C) and the last one is named (AU).

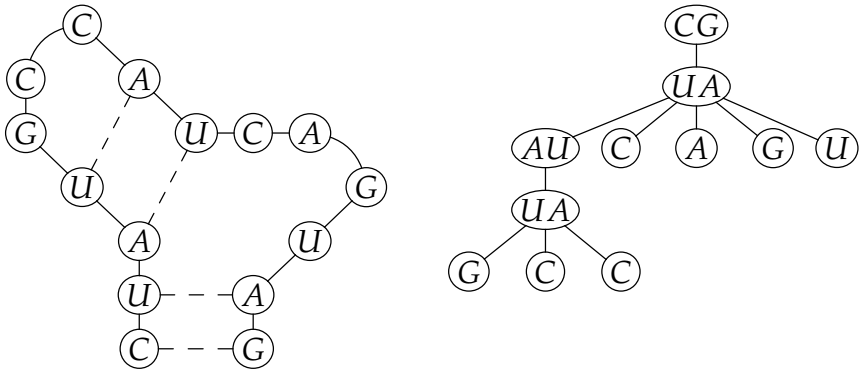
Now we continue with the further exploration. We take a hydrogen bond which we have just encountered (for example (G)–(C)) and look at the subfolding starting there. In our example, we see that the next nucleotides form a hydrogen bond (U)–(A). According to this we insert a node labelled (UA) as a child of the node labelled (GC). Last but not least, we find the three single nucleotides (A), (G) and (C) and insert them as children of the



(a) First example of RNA sequence with corresponding rooted ordered tree.



(b) Step by step transformation of an RNA folding to a rooted tree.



(c) Second example of an RNA sequence. The task is to find the tree edit distance between the two rooted trees of RNA 1 and RNA 2.

previously created node in this exact order. This is a breadth-first-search approach. One could also do a depth-first-search approach and get the same tree T in the end.
At this point, the mapping should be clear and it can be used to finish the mapped tree in Figure 1.4a respectively understand the structure of the mapped tree in Figure 1.4c.

1.3 Computing the tree edit distance

The first algorithm for computing the tree edit distance was presented by Tai [9]. It required $O(n^2_{leaves} m^2_{leaves} nm)$ time and space, so its worst case running time is $O(n^6)$. Then Sasha and Zhang [8] respectively Klein [6] developed similar algorithms using dynamic programming which improved the running time to $O(n^2 m^2)$ resp. $O(n^3 \log n)$ time. The main idea in both approaches is to find a suitable subset of *relevant subproblems*. Furthermore Dulucq and Touzet [4] proved a lower bound of $\Omega(nm \log n \log m)$ on the running time of any such dynamic programming approach.
More recently Chen [2] introduced an algorithm which makes use of fast matrix multiplication and provides a solution in $O(nm + nm^2_{leaves} + n_{leaves} m^{2.5}_{leaves})$ time and $O(n + (m + n^2_{leaves}) \min n_{leaves}, m_{height})$ space. In terms of the worst case running time, Klein provides the fastest algorithm.

In the next section, we will take a closer look at decomposition algorithm to solve the minimal tree edit distance. For that, we will see the two algorithms by Sasha and Zhang and by Klein. Afterwards, we will make some improvements and introduce a more recently published algorithm by Demaine, Mozes et al. [3], that achieves an even better running time in Section 3.

2 First Results

This section will present two algorithms, on which the optimal decomposition algorithm is based on. The first one is an $O(n^2 m^2)$ algorithm by Sasha and Zhang [8], the second one the $O(m^2 n \log n)$ algorithm by Klein [6].

As mentioned above, the two algorithms discussed in this chapter use some dynamic programming approach. For a dynamic programming approach one has to know how to branch a problem into smaller subproblems. Since we only consider deletion operations, a subproblem is obviously a subforest of the original tree F. Therefore we have a trivial number of subproblems of 2^{n+m}. Of course one can reduce this number by some simple arguments to result in a polynomial number of relevant subproblems. The relevant subproblems are those which are computed during computing $\delta(F, G)$. Furthermore a trivial subproblem is given if we consider the empty subforest of F.
As described above, the nodes have a post-order index. So suppose F consists of the set $\{1, 2, ..., n\}$ and G of the set $\{1, 2, ..., m\}$. The branching in both presented algorithm take place by either deleting the root of the left- or rightmost tree or by matching the outermost trees and the rest of the forest respectively. The choice is according to some decomposition strategy S defined as follows:

Let A be a decomposition algorithm to determine $\delta(F, G)$ using a dynamic programming approach and S its decomposition strategy. Let $F'$and G' be some subforests appearing in the computation of $\delta(F, G)$. $S(F', G') \in$ {left, right} and decides whether we compare the root of the leftmost trees of F' and G' with each other or the roots of the rightmost trees respectively. This will be explained further later on in this section. This strategy however makes sure that each possible relevant subproblem is made of a subset $\{i_F, i_F + 1, ..., j_F\}$ of $\{1, 2, ..., n\}$ for F and $\{i_G, i_G + 1, ..., j_G\}$ of $\{1, 2, ..., m\}$ for G. Therefore the number of possible relevant subproblems can be reduced to all subset of this form. This number is obviously $\binom{n}{2} = O(n^2)$ and $\binom{m}{2} = O(m^2)$ for F and G respectively. So the trivial simplification yields an upper bound of $O(n^2m^2)$ for the number of relevant subproblems.

2.1 Sasha and Zhang's algorithm

The basic idea for Sasha and Zhangs algorithm is to take a closer look at the rightmost trees R_F and R_G of F and G respectively. In other words $S(F', G')$ =right $\forall F', G'$ relevant subproblems of $\delta(F, G)$. In each step, there is either the possibility to match their roots or one of them has to be deleted. These are the only two options, since deletion is the only operation which changes the structure of the forests. This leads to the following lemma.

Lemma 2.1. *$\delta(F, G)$ can be computed considering $O(\min\{n_{height}, n_{leaves}\} \min\{m_{height}, m_{leaves}\} nm)$ subproblems as follows:*

1. $\delta(\emptyset, \emptyset) = 0$;
2. $\delta(F, \emptyset) = \delta(F - r_F, \emptyset) + c_{del}(r_F)$;
3. $\delta(\emptyset, G) = \delta(G - r_G, \emptyset) + c_{del}(r_G)$;
4. $\delta(F, G) = \min \begin{cases} \delta(F - r_F, G) + c_{del}(r_F) \\ \delta(F, G - r_G) + c_{del}(r_G) \\ \delta(R_F^\circ, R_G^\circ) + \delta(F - R_F, G - R_G) + c_{match}(r_F, r_G) \end{cases}$

First of all, let's concentrate on the correctness of the dynamic program. Constraint 1 is trivial. Moreover if there are nodes in exactly one tree, then one receives the tree edit distance by deleting all nodes in this tree. One possibility to do that is to always delete the root of the rightmost tree in the non empty tree, which is exactly what happens when one incrementally applies this constraints 2 or 3. For the 4^{th} constraint assume that F and G have some nodes. So both have a rightmost tree. There are two possibilities:

1. in the cheapest mapping r_F and r_G are matched
2. in the cheapest mapping r_F and r_G are not matched

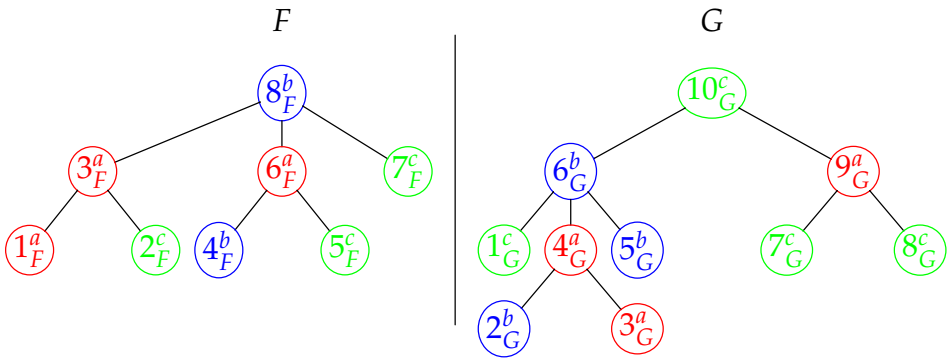
For case 1, the tree edit distance consists of the costs of matching r_F to r_G, matching the rest of the rightmost trees of F and G ($\delta(R_F^\circ, R_G^\circ)$) and last but not least matching the rest of the nodes of F and G which are not in their rightmost trees ($F - R_F, G - R_G$). This is exactly the sum given in the 3^{rd} expression of the minimum value for $\delta(F, G)$. In case 2, one of the two roots has to be deleted. These are obviously the other two expression in this minimum.

Before we go on with the prove of the running time, let us take a look at a small example to demonstrate the dynamic program. The alphabet $\Sigma = \{a, b, c\}$ consists of 3 letters. The following table defines the costs $c_{match}(\tau_1, \tau_2)$:

	a	b	c
a	0	2	3
b	2	0	1
c	3	1	0

Furthermore, the cost of deleting any node $c_{del}(\tau) = 3 \forall \tau \in \{a, b, c, \}$. For a better overview, every label is associated with a certain color.

a is red, b is blue and c is green. Now let's have a look at the two trees F and G:



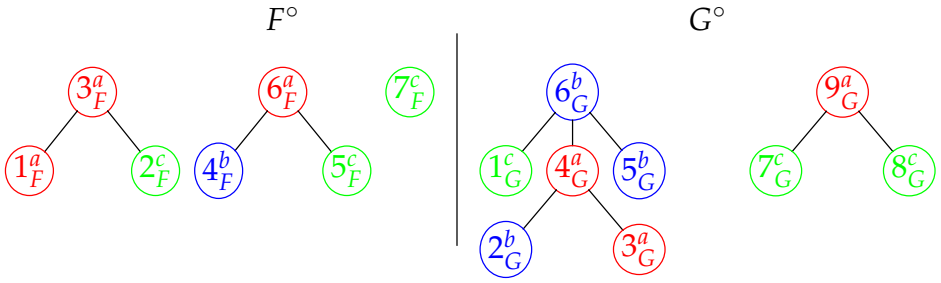
Before we start with the programm, let's have a short look at some obvious facts:

- Deletion is expensive. Relabeling is always at least as cheap as a deletion. In all other cases than matching a node with label a to another one with label c, it is strictly cheaper to relabel.
- Relabeling is symmetric. It doesn't matter whether you relabel a node with label b to have the label c or the other way around. This obviously results in a large number of different solutions with the same costs.

Now lets start with the step by step guide through the dynamic programm:
Step 1:
F and G both contain nodes and are trees. Therefore the rightmost trees are in both cases the complete trees themselves. So we have to compare the roots of F and G which are 8_F and 10_G. We can either match them (this costs just 1) or delete any of those two, which costs 3. After some investigation, it is clear that matching them is obviously the best choice.

$$=> \delta(F,G) = \delta(F^\circ, G^\circ) + c_{match}(b,c) = \delta(F^\circ, G^\circ) + 1$$

Step 2:

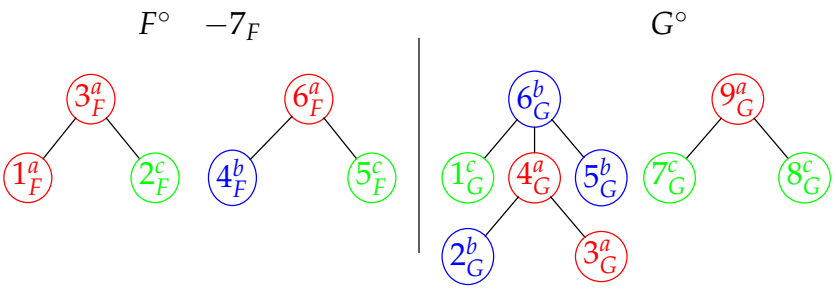


The roots of the rightmost trees in F° and G° are 7_F and 4_G. In this case we have the same costs whether we delete one node or match them. So we have to take a closer look at the three possibilities that we have, especially on the minimum number of nodes that we have to delete in all three possible branches. *Step 2a:* match 7_F and 4_G.
In the following progression, we have to delete the children of 4_G (as 7_F doesn't have any children) and at least one of the two other subtrees of F°. So without the last steps of mapping L_{G° and the resulting subtree of F° we have to delete at least 5 nodes.
Step 2b: deleting 4_G.
The resulting subforest of G has two single nodes, which means that we have to map them to a subforest of F° in some kind. That demands a deletion of at least 2 nodes in F°, so all in all a deletion of 3 nodes (once again without considering the mapping of L_{G°.
Step 2c: deleting 7_F.
This is obviously the best choice. Expect for the mapping of L_{G° we don't have to delete any further nodes. In addition to that, the subtrees starting at 6_F and 4_G already look very much alike.
So we go on with deleting 7_G.

$$=> \delta(F^\circ, G^\circ) = \delta(F^\circ - 7_F, G^\circ) + c_{del}(7_F) = \delta(F^\circ - 7_F, G^\circ) + 3$$

$$=> \delta(F, G) = \delta(F^\circ - 7_F, G^\circ) + 4$$

Step 3:



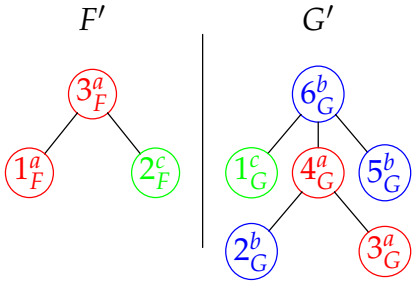
The rightmost roots are 6_F and 9_G. Both of them are labelled a, so it is trivial that we should match those two nodes. After this step, we will match 5_F and 8_G since both of them have the label c. Last but not least, it is obviously the cheapest to match 4_F and 7_G and just relabel one of them. If we define the subtree of F° rooted at 3_F as F' and the subtree of G° rooted

at 6_G as G', then the following equality holds:

$$\delta(F^\circ - 7_F, G^\circ) = \delta(F', G') + c_{match}(b,c) + c_{match}(a,a) + c_{match}(c,c) = \delta(F', G') + 1$$

$$=> \delta(F, G) = \delta(F', G') + 5$$

Step 5:



It is trivial that we have to delete at least three nodes from G' and that the cheapest solution will delete exactly three nodes. The easiest way to go on is to make a case destinction.
Step 5a: matching 3_F and 6_G.
$c_{match}(a,b) = 2$. If we go on, we immediately see, that we have to relabel at least one additional node. The lower bound for the costs of relabelling is 1. This lower bound can be achieved if we match 1_F with 4_G (or equivalently 3_G after deleting 4_G) and 2_F with 5_G, since we only have to change the label of 2_F or 5_G respectively. Of course we have to delete the other three nodes. All in all, this case would cost $3 + 3 * 3 = 12$.
Step 5b: deleting 6_G.
G'° consists of two single nodes and the subtree rooted at 4_G. We have the possibility to delete 3_F or match it to one of the nodes in G'°. If we delete it, then we have left only two nodes in F'° and four nodes in G'°. Therefore we have to delete at least four nodes overall, leading to a mapping which costs ≥ 12. That means that we cannot improve our previous result, so we can forget about this possibility. Furthermore, if we map 3_F to any other node than 4_G, we have to delete both children of 3_F since no other node has any children other than 4_G. This would also not lead to an improvement. So the only possibility left is to match 3_F to 4_G, 1_F to 2_G and 2_F to 3_G. The costs for this possibility are $0 + 2 + 3 + 3 * 3 = 14$. To conclude: deleting 6_G doesn't give us an improvement.

$$=> \delta(F', G') = 12$$

$$=> \delta(F, G) = \delta(F', G') + 5 = 12 + 5 = 17$$

This means the tree edit distance between F and G is 17. The computation above was based on comparing different branching possibilities. The algorithm of Sasha and Zhang would compute any possible combination of relevant subproblems and would yield the same result as we figured out above.

This small example shall give an idea on how Sasha and Zhang's algorithm works. This will help to understand the further investigation on the running time.
First of all, let's prove the statement which has already been given in the beginning of this section: The number of relevant subproblems is $O(n^2m^2)$. To prove that, we enumerate the nodes in both trees according to their postorder index. Observe:

Claim 2.2. $r_{F'}$ and $r_{G'}$, the roots of the rightmost trees for some relevant subforests F' and G', always have the highest index.

Proof. This can be checked via an induction. The claim is correct for F and G due to the order. Suppose this holds for some F' and G' and consider all relevant suforests F'' for F' (G' is analogous). These are $F' - r_{F'}$, $F - R_{F'}$ and $R^{\circ}_{F'}$. If we take a look at the children of $r_{F'}$, we see that the rightmost child of $r_{F'}$ has the second highest index of F' right after $r_{F'}$. At the same time, the rightmost child is the root of the rightmost tree in the forest after deleting $r_{F'}$. In the case that $r_{F'}$ doesn't have any children, the next root would be the left sibling of it, which obviously is the node with the highest index in the remaining tree. This concludes the argumentation for $F' - r_{F'}$ and $R^{\circ}_{F'}$. If we disregard the whole rightmost tree, we see that the node with the highest index is the left sibling of $r_{F'}$ once again. In $F - R_{F'}$ this node is the root of the rightmost tree, finishing our argument. □

Claim 2.3. All subtrees considered in the dynamic program are made of a subset $\{i_F, ..., j_F\}$ for F (and for G analogously).

Proof. Once again we make an induction and thus take a look at the branching process. At the beginning $i_F = 1$ and $j_F = n$, forming our induction basis. suppose the induction hypothesis holds for an F' within the dynamic program. The possible relevant subforests are $F' - r_{F'}$, $F' - R_{F'}$ and $R^{\circ}_{F'}$. By the statement above, $F' - r_{F'}$ consists of the set $\{i_{F'}, ..., j_{F'} - 1\}$. As we argued above, the left sibling of $r_{F'}$ has the highest index in $F - R_{F'}$ denoted by k. This means that $\{k + 1, ..., j_{F'}\} \subseteq R_{F'}$. Due to the indexing itself, it is

clear that this indeed is an equality. This concludes that $\{k+1, ..., j_{F'}\} = R_{F'}$ and $\{i_{F'}, ..., k\} = F' - R_{F'}$, proving the correctness of the induction step. □

Now we prove the smaller running time of $O(\min\{n_{height}, n_{leaves}\} \min\{m_{height}, m_{leaves}\} nm)$. We calculate the number of relevant subproblems independently for both trees F and G and multiply them afterwards. This is obviously an upper bound for the relevant subproblems which appear in the calculation of $\delta(F, G)$.
First of all, all forests of the form F_v° for some $v \in F$ are relevant subproblems. There has to be a branch in the dynamic program, where all nodes on the right of v have been deleted. Then v is per definition the root of the rightmost tree in this (relevant) subproblem. After matching or deleting v, F_v° becomes a relevant subproblem.
For further investigations, we need two concepts. The first one is the concept of prefixes. We say that a relevant supbroblem F' of F is a prefix of another (relevant) subproblem F'' of F if the following holds for $i_{F'}$,$j_{F'}$,$i_{F''}$,$j_{F''}$ as defined in Claim 2.3:

$$i_{F'} = i_{F''} \qquad \text{and} \qquad j_{F'} \leq j_{F''}$$

The second one is Sasha and Zhang's definition of keyroots:

$$\text{keyroots}(F) := \{\text{root of } F\} \cup \{v \in F \mid v \text{ has a left sibling}\}$$

The role of these keyroots is that any relevant subproblem of F is a prefix of some F_v° where $v \in \text{keyroots}(F)$. Suppose there is some relevant subproblem F' given consisting of nodes $\{i_1, ..., j_1\}$. Then either $i_1 = 1$ or $i_1 > 1$. In the first case, under the assumption that $j_1 \neq n$, F' is a prefix of $F^\circ = F_r^\circ$ itself, where r is the root of F. In the other case, due to the ordering of F, there must be some subtree which lies completely on the left of F'. Therefore the root of the biggest tree on the left is the left sibling of the root of the tree which contains F' as a subforest. This shows that F' is the prefix of the subtree rooted at the right sibling, which is a keyroot by definition. In Figure 2.1 the set $\{7, 8\}$ is an example of a set, which is not equal to F_v° for any $v \in F$ but is a prefix of $F_{11}^\circ = \{7, 8, 9, 10\}$. It is a relevant subproblem, because after deleting the nodes 13,12,11,10 and 9 this set will appear in the dynamic program.

Furthermore, they defined the collapse depth $\text{cdepth}(v)$ to be the number of keyroot ancestor of v and $\text{cdepth}(F) := \max\{\text{cdepth}(v) \mid v \in F\}$ being

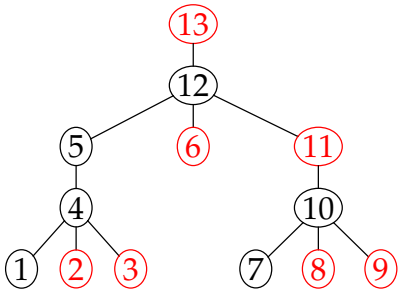


Figure 2.1: Illustration of the keyroots for Figure 1.4a as an example. The numbers show the post-order indexing.

the maximal cdepth. With this definition, the following chain of relations is rather trivial:

$$\sum_{v \in \text{keyroots}(F)} |F_v^\circ| = \sum_{v \in F} \text{cdepth}(v) \leq \sum_{v \in F} \text{cdepth}(F) = |F|\text{cdepth}(F). \quad (2.1)$$

Sasha and Zhang furthermore proved, that $\text{cdepth}(F) \leq \min\{n_{height}, n_{leaves}\}$. Therefore there are $\leq n \min\{n_{height}, n_{leaves}\}$ relevant subproblems for F, and equivalently $\leq m \min\{m_{height}, m_{leaves}\}$ relevant subproblems for G. This concludes the proof of the running time of their algorithm of $O(\min\{n_{height}, n_{leaves}\} \min\{m_{height}, m_{leaves}\} nm)$.

2.2 Klein's algorithm

Klein improved the algorithm of Sasha and Zhang to yield a running time of $O(n^3 \log n)$. The only difference between those two algorithms lies in the choice of the subtrees used in step 4 of the dynamic programming in Lemma 2.1. Sasha and Zhang always take the rightmost tree and make the next recursive step. Klein on the other hand decides this direction by comparing the sizes of the outermost trees. If $|V(L_F)| > |V(R_F)|$, the algorithm performs just like Sasha and Zhang's. However, if $|V(L_F)| \leq |V(R_F)|$, he performs the recursion on the leftmost tree L_F. So $S(F', G')$=left if and only if $|V(L_{F'})| \leq |V(F')|$ for every F', G' relevant subproblems of $\delta(F, G)$. Changing this seemingly small detail indeed improves the worst case running time to $O(n^3 \log n)$. One proof for this result uses the so called *heavy path decomposition* which was introduced by Harel and Tarjan [5]. The definition is a recursive one. We mark the root of a tree as *light* and afterwards

choose one child among those with the most descendants arbitrarily to be marked as *heavy* while the other children get marked as light. We continue this way for every node until every node is marked as either light or heavy. We call an edge heavy, if it connects a non-leaf node with its heavy child. We call a path which connects a light node with a leaf and only consists of heavy edges to be a heavy path. Leaf-nodes which are light are a special case of a heavy path of length 0.

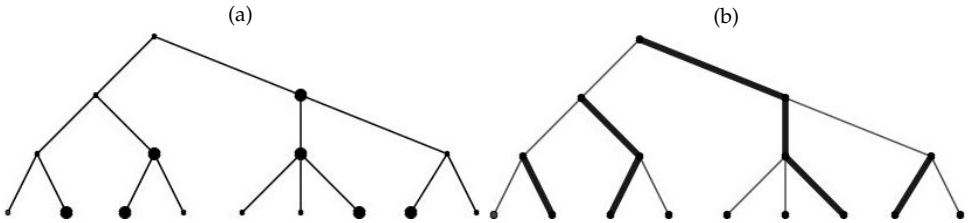


Figure 2.2: Example of a heavy path decomposition. Figure a) shows the heavy nodes, Figure b) the correspondig heavy paths.

After having fixed such a heavy path decomposition, one can define the $\mathrm{ldepth}(v)$ to be the number of proper ancestors of v which are marked as light. Once again we extend this definition such that $\mathrm{ldepth}(F) = \max\{\mathrm{ldepth}(v) \mid v \in F\}$. One can show, that $\mathrm{ldepth}(v) \leq \log(F) + O(1)$. The key idea now is, that every relevant subforest of F is obtained by some $i \leq |F_v|$ consecutive deletions from F_v for some light node v. With a very similar computation as in the equation system 2.1.

In Figure 2.3 the different choice of directions in both algorithms are illustrated. Of course, the dynamic program would split the subproblems in many steps. For demonstrating purposes, we ignore this and show the whole tree which hasn't been deleted yet.

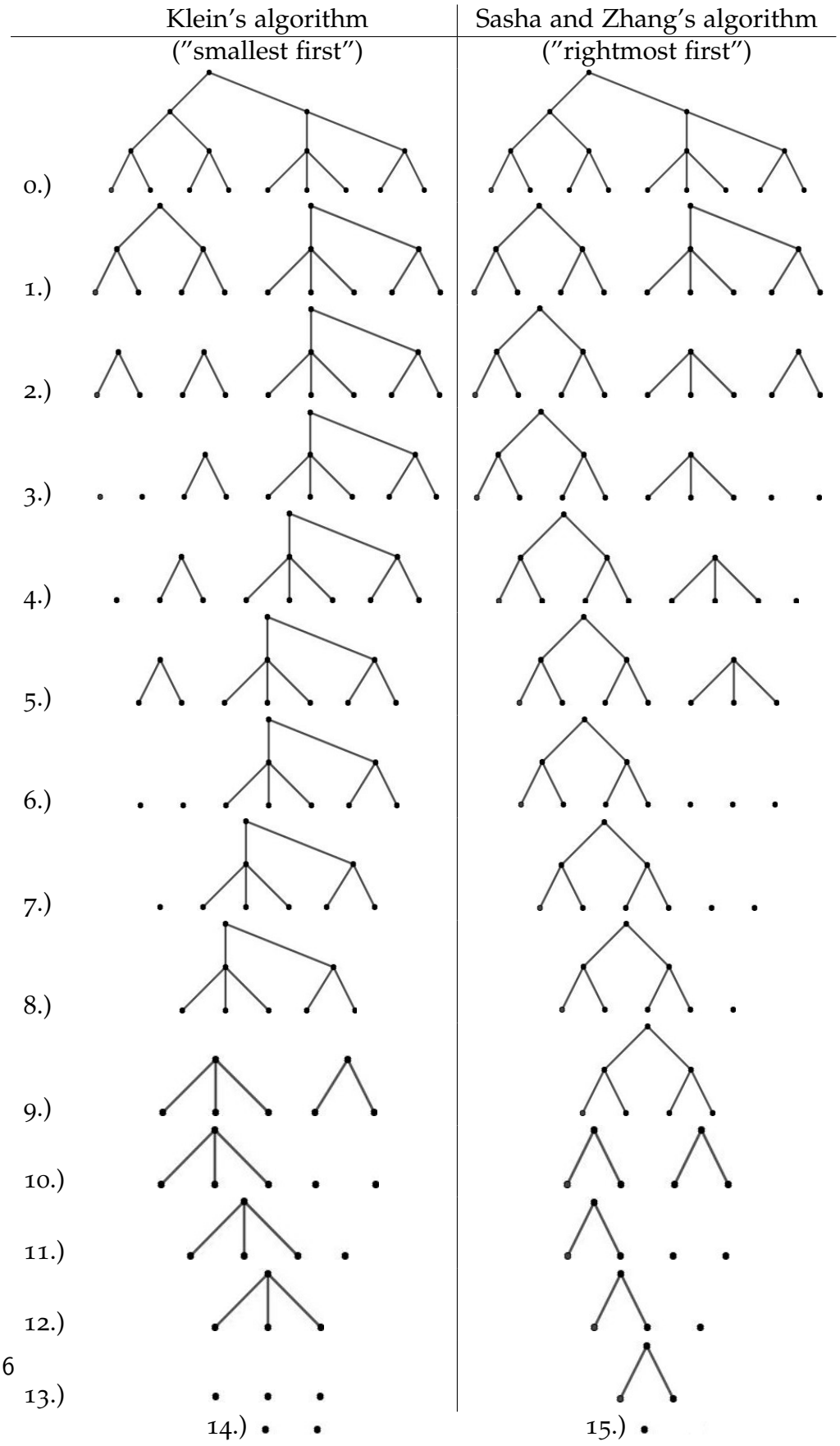


Figure 2.3: Comparison of the recursive choice of direction.

3 An optimal Algorithm

In this section we will present the Demaine, Mozes et al's implementation of an optimal algorithm for computing $\delta(F, G)$. Suppose we have a heavy path decomposition for a tree F given. We call the unique heavy path of F which originates in the root of F to be the main heavy path. We define the set TopLight(F) to be all light nodes with ldepth 1:

$$\text{TopLight}(F) = \{v \mid \text{ldepth}(v) = 1 \text{ and } v \notin \text{the main heavy path of } F\}$$

So a light node $v \in F$ is in TopLight(F) if and only if its parent lies on the main heavy path of F. If we compute $\delta(F_v, G)$ for all $v \in \text{TopLight}(F)$ we compute all relevant subproblems $\delta(F_v^\circ, G_w^\circ)$ for any $v \in F$, $w \in G$ not on the heavy path as well. This has been proven when we counted the number of relevant subproblems earlier.

3.1 The Algorithm

We compute $\delta(F, G)$ recursively as follows:

1. If $|F| < |G|$, compute $\delta(G, F)$ instead.
2. Recursively compute $\delta(F_v, G)$ $\forall v \in \text{TopLight}(F)$ using these recursive steps.
3. Compute $\delta(F, G)$ using the following decomposition strategy S:
   $S(F', G') =$ left if F' is a tree, or if $l_{F'}$ is not the heavy child of its parent. Otherwise $S(F', G') =$ right. However, do not recurse into subproblems that were previously computed in step 2.

3.1.1 Correctness

The first step makes sure that F is the larger forest. Step 2 ensures that $\delta(F_{v'}^\circ, G_w^\circ)$ is computed for all v' not in the heavy path of F and all $w \in G$.

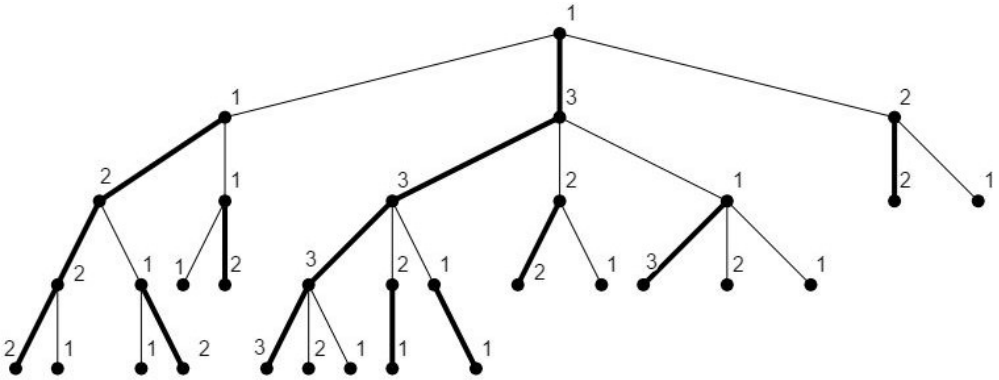


Figure 3.1: The number at each node indicates the order among siblings in which they are considered in the algorithm.

Step 3 is an implementation of Klein's decomposition strategy if we only had binary trees. For other trees, it always performs the left deletions first until the leftmost tree is the heavy child. Then it goes on with all the right deletions. Therefore this strategy takes the heavy child always at last. This implementation is obviously a decomposition strategy, wherefore the correctness is due to the correctness of the general concept of such strategies. We showed the correctness for the special case of $S(F, G) = \text{right}$ in Section 2.1

Before we continue with the time complexity, we want to present a short example of how the algorithm chooses its decomposition strategy. As the caption indicates, the number at each node indicates some order among children. For further explanation take a look at some $v \in F$. The children of v are labelled from 1 to 3 (or some subset). We call them v_i where i is the nodes label. Note that the heavy child of some node always has the highest index among its siblings. Suppose that in step 3 of the algorithm we want to compute some $\delta(F_v^\circ, G')$ for some G' being a relevant subproblem of G. If $\mathrm{S}(F_v^\circ, G') = \text{left}$, then v_1 is on the left side of the heavy node. If $\mathrm{S}(F_v^\circ, G') = \text{right}$, then the leftmost root has to be the heavy child of v due to the decomposition strategy S. If we go on to perform this strategy on the pair $(F_v^\circ - F_{v_1}, G')$, we encounter v_2 the same way as v_1 above.

3.1.2 Time Complexity

We want to show a worst case running time of $O(m^2 n(1 + \log \frac{n}{m}))$. We define the trivial subproblems to be those, where at least one of the two forests is empty, i.e. every problem $\delta(F, G)$ where F or G is equal to the empty graph. If we reach a trivial subproblem, we can calculate the costs immediately by summing up the costs of deleting all remaining nodes. All other subproblems are obviously harder to solve and therefore not trivial. All in all there are $O(n^2)$ distinct trivial subproblems.
We define $\mathrm{R}(F, G)$ to be the number of distinct non-trivial subproblems which appear in the dynamic programming approach of computing $\delta(F, G)$. Notice first that the following two properties hold:

$(*)$ $\sum_{v \in \mathrm{TopLight}(F)} |F_v| \leq |F|$. Because F_v and $F_{v'}$ are disjoint for all $v, v' \in \mathrm{TopLight}(F)$.

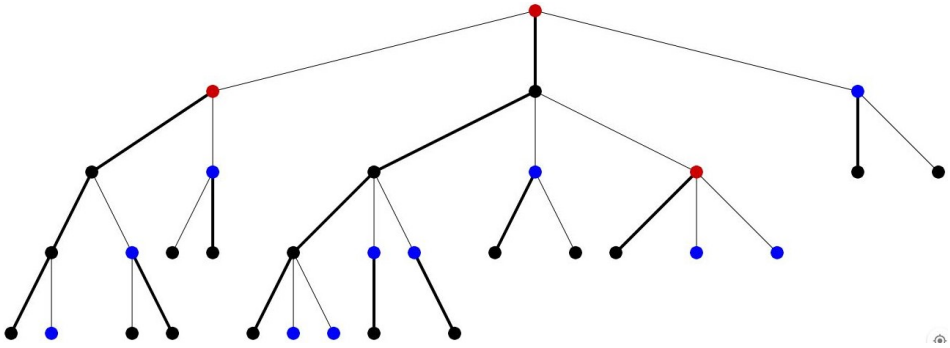
$(**)$ $|F_v| < \frac{|F|}{2}$ $\forall v \in \mathrm{TopLight}(F)$. Otherwise v would have been a heavy node.

Notice moreover that when computing the sum $\sum_{v \in \mathrm{TopLight}(F)} R(F_v, G)$, we already compute all possible relevant subproblems of matching some node v not on the heavy path of F with any node $w \in G$. Now we have to bound the number of remaining subproblems. We can bound the number of relevant subproblems by multiplying the ones of in F and G. For G, we just take all $O(|G|^2)$ possible subproblems (via left and right deletions). Furthermore, note that for a node v on the main heavy path of F, v can only be matched or deleted if the remaining subforest is exactly F_v. In this case, it is irrelevant whether we match v or whether we delete it, both result in the new relevant subforest F_v°. Therefore it suffices to consider only the $|F|$ deletions according to the strategy. Thus the total number of relevant subproblems is bounded by $|G|^2|F|$. Therefore if w.l.o.g. $|F| \geq |G|$ then

$$R(F, G) \leq |G|^2|F| + \sum_{v \in \mathrm{TopLight}(F)} R(F_v, G) \tag{3.1}$$

Lemma 3.1. $R(F, G) \leq 4(|F||G|)^{3/2}$

Proof. We perform an induction on $|F| + |G|$. The induction base of $|F| = |G| = 0$ is trivial since both trees are empty, so $R(F, G) = 0$. For the



induction step, we use the inequality (3.1):

$$
\begin{aligned}
R(F,G) &\leq |G|^2|F| + \sum_{v\in \text{TopLight}(F)} R(F_v,G) \\
&\overset{\text{Lemma 3.1}}{\leq} |G|^2|F| + \sum_{v\in \text{TopLight}(F)} 4(|F_v||G|)^{3/2} \\
&= |G|^2|F| + 4|G|^{3/2} \sum_{v\in \text{TopLight}(F)} |F_v|^{3/2} \\
&\leq |G|^2|F| + 4|G|^{3/2} \sum_{v\in \text{TopLight}(F)} |F_v| \max_{v\in \text{TopLight}(F)} \sqrt{|F_v|} \\
&\overset{(***)}{\leq} |G|^2|F| + 4|G|^{3/2}|F|\sqrt{\frac{|F|}{2}} \\
&= |G|^2|F| + \sqrt{8}(|F||G|)^{3/2} \leq 4(|F||G|)^{3/2}.
\end{aligned}
$$

In step $(***)$ we used the facts $(*)$ and $(**)$. The case where $|F| < |G|$ is symmetric. □

If one takes a closer look on the structure of the subproblems created and for which nodes recursive calls are made, one can show the actual bound of $O(m^2 n(1 + \log \frac{n}{m})$. Take a look at the following two set A and B:

$$
\begin{aligned}
A &= \{a \in \text{light}(F) : |F_a| \geq m\} \\
B &= \{b \in F - A : b \in \text{TopLight}(F_a) \text{for some } a \in A\}
\end{aligned}
$$

First we give a short illustration of these sets A and B. Suppose G is already quite small such that $m = 4$. F is the tree in figure **??**. Nodes in set A are colored red and nodes in set B are colored blue.
One can show, that the nodes in $A \cup B$ are exactly those, for which recursive calls are made using the whole tree G. All relevant subproblems considered

in the computation of $\delta(F, G)$ are either created in a recursive call $\delta(F_a, G)$ for some $a \in A$ or in the computation of $\delta(F_b, G)$ for some $b \in B$. Using the previously found bound and some additional conditions like the fact $(**)$, one gets the bound of $O(m^2 n(1 + \log \frac{n}{m})$.

3.2 Lower Bound

The algortihm presented above actually achieves the lower bound for decomposition algorithms. The authors proofed this tight lower bound of $O(m^2 n(1 + \log \frac{n}{m})$ in. We will show a more relaxed lower bound of $\Omega(m^2 n)$ in the following paragraphs.

Lemma 3.2. *For any decomposition algorithm solving the tree edit distance problem, there exists a pair of trees (F, G) with sizes $n > m$ resprectively, such that the number of relevant subproblems is $\Omega(m^2 n)$*

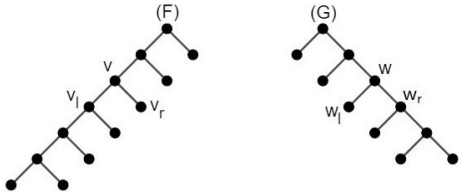


Figure 3.2: Sketch of F and G which fulfill a lower bound on the running time for each decomposition algorithm of $\Omega(m^2 n)$

Proof. Let S be the strategy of decomposition algorithm and assume F and G to be as in Figure 3.2. As previously stated, every pair (F_v°, G_w°) for $v \in F$, $w \in G$ is a relevant subproblem for S. We count the number of such relevant subproblems, where v and w are inner nodes of F and G. For an inner node x, let us denote x's left child as x_l and x's right childs as x_r. In the forest F_v° the rightmost root is v_r and in G_w° the leftmost root is w_l. In each step, the strategy chooses the direction from which side we should delete. Every time the strategy chooses left, we delete from F and otherwise from G. This computational approach always keeps v_r as rightmost and w_l as leftmost roots of their respective forests until they are the only nodes left. So it takes at least $\min\{|F_v|, |G_w|\} - 1$ steps until every relevant subproblem of (F_v°, G_w°) is found. Since v_r and w_l are the outermost roots, the computational paths

of (F_v°, G_w°) and $(F_{v'}^\circ, G_{w'}^\circ)$ are completely disjoint. Because of their structure, there are $\frac{n}{2}$ respectively $\frac{m}{2}$ internal nodes resulting in the following equation:

$$\sum_{(v,w)\text{internal nodes}} \min\{|F_v|, |G_w|\} - 1 = \sum_{i=1}^{\frac{n}{2}} \sum_{j=1}^{\frac{m}{2}} \min 2i, 2j = \Omega(m^2 n).$$

□

If $m \neq \Theta(n)$ then this bound doesn't match the running time of the algorithm, since the lower bound should be $\Omega(m^2 n \log \frac{n}{m})$. In this case, one can prove that using trees like in Figure 3.3 results in a running time of $O(m^2 n \log \frac{n}{m})$ for every decomposition strategy.

4 Conclusion

This report provides a short insight in a rather modern topic of the tree edit distance.

First, we introduce the problem and define some basic notations. We gave an example on how an RNA folding can be mapped to a rooted tree, such that our results can be used in the context of TNA foldings..

We proceeded to give two quick dynamic programming algorithms which differ only in the choice of decomposition strategy. To analyze the running time, we counted the relevant subproblems which are considered in the recursive computation. We use the concepts of keyroots and the heavy nodes to conclude the running times of $O(m^2 n^2)$ and $O(m^2 n \log n)$ respectively.

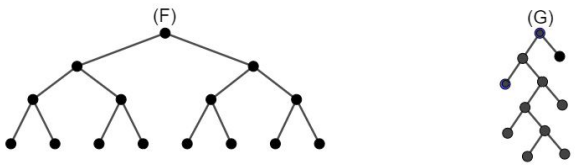


Figure 3.3: F and G used to prove a lower bound of $\Omega(m^2 n \log \frac{n}{m})$

In Section 3 we present the algorithm of Demaine, Mozes et al., which improves the running time of Klein's algorithm even further. Once again only the choice of direction within the dynamic program is adjusted. For the effectiveness of the algorithm, the special structure of nodes in the set TopLight(F) is exploited. We give a sketch of the calculations to obtain a running time of $O(m^2 n(\log \frac{n}{m}))$. Last but not least we show that this is also a lower bound for all decomposition strategies. This implies that the presented algorithm is an efficient approach to compute the tree edit distance.

Bibliography

[1] Bogdanowicz D., Giaro K. and Wróbel B., *TreeCmp: Comparison of Trees in Polynomial Time*, Evolutionary Bioinformatics 8, 2012, 475–487

[2] Chen W., *New algorithm for ordered tree-to-tree correction problem*, J. Algor. 40, 2001, 135–158

[3] Demaine E. D., Mozes S., Rossmann B. and Weimann O., *An Optimal Decomposition Algorithm for Tree Edit Distance*, ICALP'07 Proceedings of the 34th international conference on Automata, Languages and Programming, 2007, 146–157

[4] Dulucq S. and Touzet H., *Analysis of tree edit distance algorithms*, Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM), 2003, 83–95

[5] Harel D. and Tarjan R. E., *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput. 13, 2, 1984, 338–355

[6] Klein P. N., *Computing the edit-distance between unrooted trees*, Proceedings of the 6th Annual European Symosium on Algorithms (ESA), 1998, 91–102

[7] Moore P., *Strucural motifs in RNA*, Ann. Rev. Biochem 68, 1999, 287–300

[8] Sasha D. and Zhang K., *Simple fast for editin distance between trees and related problems*, SIAM J. Comput. 18, 6, 1989, 1245–1262

[9] Tai K., *The tree-to-tree correction problem*, J. Assoc. Comp. Mach. 26, 1979, 422–433

[10] Wagner R. and Fischer M. J., *The string-to-string correction problem*, J. ACM 21, 1, 1974, 168–173