Clemens Andritsch

Comparing Trees

Master's Thesis

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 2020

Contents

1 Introduction

Different areas of research have to compare trees at some point. Bioinformatics need a measure for comparing the similarities between different RNA-structures and computer scientists have to compare strucured text databases or natural languages. These are just three example of a wide range of real life applications of tree distances.

This thesis shall give a short overview of three widely used tree comparison techniques. It explains the advantages and disadvantages of these techniques. Furthermore we present an implementation of one of these techniques and discuss the results.

2 Basics and notation

In this chapter we introduce some basic notions that we shall use throughout this Thesis. They include special property of trees, basic concepts for the nodes in trees and more evolved notations that will be needed in later section of this Thesis

2.1 Basic Graph Theoretic Concepts

Definition 2.1. Let $T = (V, E)$ be a (simple) graph. T is called a tree if it is connected and acyclic, meaning that for any pair of vertices $v \neq w \in V$ there exists exactly one path that has v and w as its endpoints. T is called *rooted* if one node $r \in V$ is designated as the root of the tree. T is a *labelled* tree, if there exists a labelling function $l : \ V \mapsto \Sigma$, where Σ is an arbitrary set of labels.

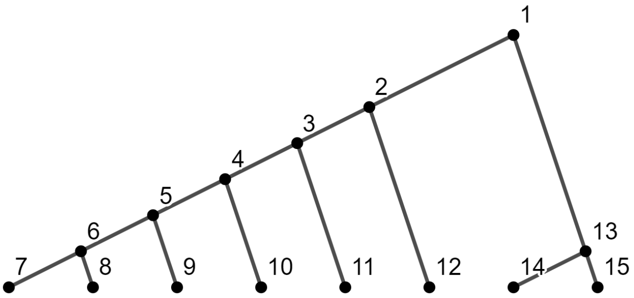


Figure 2.1: The illustration suggests, that the node 1 is the root of the tree. Together with some function $l : \ \{1, ..15\} \mapsto \{A, B, C\}$ T is a rooted labelled tree.

Definition 2.2. Let $T = (V, E)$ be a rooted labelled tree with root $r \in V$. We define the *parent* $P(v) \in V$ of a node $v \in V \setminus \{r\}$ to be direct predecessor of v on the unique path from r to v in T. The parent of r is undefined.

Definition 2.3. Let $T = (V, E)$ be a rooted labelled tree with root $r \in V$, let $w, v \in V$. The node w is a *child* of v if and only if v is the parent of w. We denote by $C_T(v)$ the *set of all children* of v:

$$C_T(v) := \{w \in V | P(w) = v\}.$$

If the setting is clear one can use the shortcut $C(v)$ for $C_T(v)$

Definition 2.4. Let $T = (V, E)$ be a rooted labelled tree with root $r \in V$, let $w \neq v \in V \setminus \{r\}$. The node w is a sibling of v if and only if $P(v) = P(w)$. We denote by $S(v)$ the *sibling group of v*:

$$S(v) := \{w \in V | P(w) = P(v)\}.$$

For the special case of the root r we define the sibling group manually by $S(r) := \{r\}$.

Remark. Note that for a node v the following inclusion holds naturally:

$$v \in S(v)$$

That also implies that $|S(v)| >= 1$.

Definition 2.5. Let $T = (V, E)$ be a rooted tree. We call T *ordered* if all siblings have a specific and fixed order among each other.

Definition 2.6. Let $T = (V, E)$ be a rooted ordered tree with root $r \in V$ and let $n := |V|$. The *post-order index* is a way of enumerating the nodes of T from 1 to n. For that, you perform the following routine recursively starting with $v = r$ and index $m = 1$:

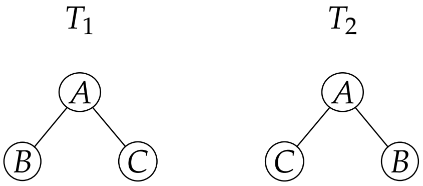


Figure 2.2: If we assume that T_1 and T_2 are unordered trees, then $T_1 = T_2$. But if we consider them to be ordered from left to right as in the figure, then $T_1 \neq T_2$.

Algorithm 1 Assign the *post-order* index to a tree T

```
function ROUTINE($v$, $m$)
    if ($v$ is a leave) or (all children of $v$ are indexed) then
        Index $v$ with the index $m$;
        ROUTINE($P(v)$, $m + 1$)
    else
        Let $w$ be the left-most child of $v$ that has not yet been indexed.
        ROUTINE($w$,$m$);
    end if
end function
```

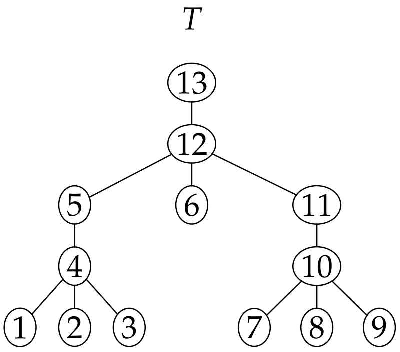


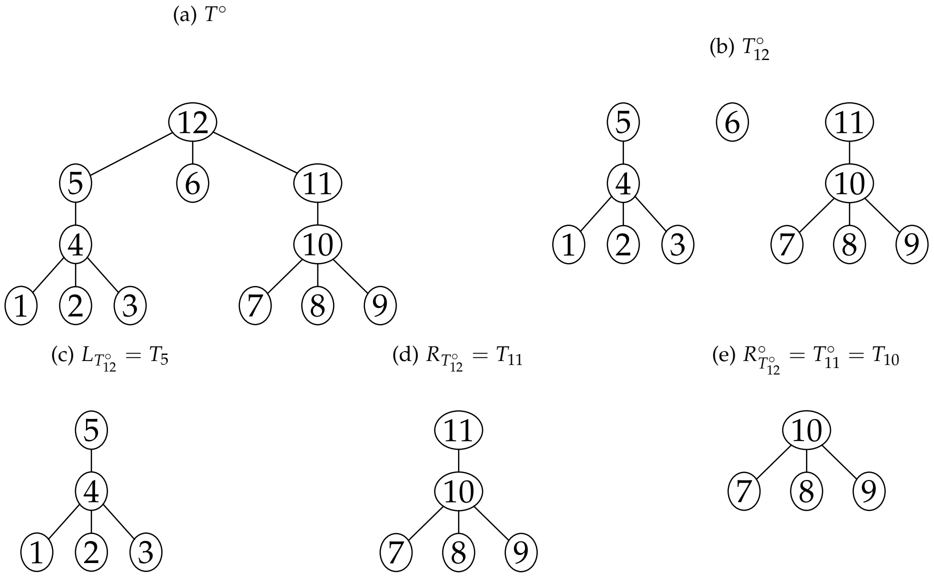
Figure 2.3: An example of the post order indexing. Note that for every subtree the root is indexed at last.

Definition 2.7. Let $T = (V, E)$ be an ordered tree rooted at $r \in V$. If $|V| > 1$ we denote by $T^\circ := T \setminus \{r\}$ be the forest, which results from T after deleting the root r. Moreover for a node $v \in V$ we denote by F_v the subtree of T rooted at v.

Definition 2.8. Let $T = (V, E)$ be an ordered forest. We denote by L_T and

R_T the left- and rightmost subtrees of T respectively. Furthermore the roots of L_T and R_T are denoted by l_T and r_T respectively

Remark. Consider the tree T in Figure 2.3. Then we can use the notion introduced above to describe the following trees:



(a) T°

(b) T°_{12}

(c) $L_{T^\circ_{12}} = T_5$

(d) $R_{T^\circ_{12}} = T_{11}$

(e) $R^\circ_{T^\circ_{12}} = T^\circ_{11} = T_{10}$

Figure 2.4: Different subtrees using the introduced notion.

Definition 2.9. Let $T_i = (V_i, E_i)$ be a rooted tree. We call $t_i := |V_i|$ the size of the tree T_i. Furthermore we use the notion of $t_{l,i}$ for the number of leaves in T_i and $t_{h,i}$ for the length of the longest path from the root to any leaf.

Remark. We will need this definitions mainly for stating the running times of algorithms. Since we want to compare T_1 with T_2 we need to be able to separate those numbers accordingly. Furthermore, if a tree is significantly bigger, we always assume that tree T_1 is the bigger one, i.e. $\mathcal{O}(t_1) \geq \mathcal{O}(t_2)$.

Definition 2.10. Let T be a forest which is ordered according to the post order indexing. Let T', T'' be two induced subforests of T with $\exists i_{T'}, j_{T'}, i_{T''}, j_{T''}$ s.t. $V(T') = \{i_{T'}, ..., j_{T'}\}$ and $V(T'') = \{i_{T''}, ..., j_{T''}\}$.
We call T' to a *prefix* of T'' if and only if the following holds:

$$i_{T'} = i_{T''} \quad \text{and} \quad j_{T'} \leq j_{T''}$$

Definition 2.11. Let T be an ordered tree rooted at r. Then we define the *keyroots* of T to be set of all nodes that have a left sibling:

$$\text{keyroots}(T) := \{r\} \cup \{v \in V(T) \mid v \text{ has a left sibling}\}.$$

Assume T is an ordered forest with trees $T_1, ..., T_n$ rooted at $r_1, ...r_n$. Then we define the set of keyroots as the union of the separated sets of keyroots:

$$\text{keyroots}(T) = \bigcup_{i=1}^{n} \text{keyroots}(T_i).$$

Definition 2.12. Let T be an ordered tree rooted at r. For a node v we define the *collapse depth* of v cdepth(v) to be the number of keyroot ancestors of V:

$$\text{cdepth}(v) = |\{w \in V(T) \mid w \text{ is an ancestor of } v\} \cap \text{keyroots}(T)|.$$

Definition 2.13. Let T be an ordered tree rooted at r. For every non-leaf node n we choose one node $m \in C(n)$ among those with the most descendants in $C(n)$ arbitrarily. We define m to be a heavy node. All non-heavy nodes are defined as *light*, especially the root r.

Remark. In most cases a tree has multiple possibilities for the definition of heavy nodes. For example if there are multiple leaves with the same parent.

Definition 2.14. Let T be an ordered tree rooted at r and let there be a fixed definition of heavy nodes. We call an edge to be *heavy* if it connects a non-leaf with its heavy child. Furthermore we call a path, which connects a light node with a leaf and only consists of heavy edges a heavy path. We call the heavy path originating at the root r the *main heavy path*. The set of all heavy paths is called a *heavy path decomposition*.

Remark. Light leaves are a special case of heavy paths of length 0.

Remark. The heavy path decomposition depends on the choice of heavy nodes.

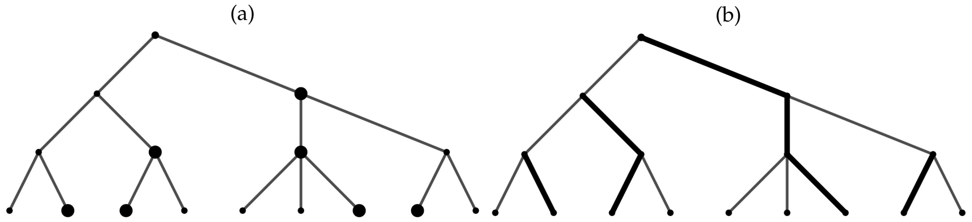


Figure 2.5: Example of a heavy path decomposition. Figure a) shows the heavy nodes, Figure b) the correspondig heavy paths.

Definition 2.15. Let T be an ordered tree rooted at r, $v \in V(T)$ and suppose a heavy path decomposition is fixed. we define the *light depth* $\text{ldepth}(v)$ to be the number of light proper ancestors of v.
Furthermore we define $\text{ldepth}(T)$ as follows:

$$\text{ldepth}(T) = \max\{\text{ldepth}(v) \mid v \in T\}.$$

Definition 2.16. Let T be an ordered tree rooted at r and let a heavy path decomposition be given. We define the set $\text{TopLight}(T)$ to be the set of all light nodes $v \in V$ with $\text{ldepth}(v) = 1$:

$$\text{TopLight}(T) := \{v \mid \text{ldepth}(v) = 1 \text{ and } v \text{ not in the main heavy path of } T\}.$$

Remark. A light node $v \in V(T)$ is in $\text{TopLight}(T)$ if and only if its parent lies on the main heavy path of T.

Definition 2.17. Let T be a tree, X the set of leaves of T and Σ a set of labels. Then T is an *unrooted phylogenetic* tree if all leaves are labelled bijectively with some label in Σ, all interior leaves are unlabelled and all interior nodes have degree at least three. A *rooted phylogenetic* tree is a phylogenetic tree where one node, the root $r \in V(T)$, is distinguished from the others and may have degree two.

Definition 2.18. Let T be a phylogenetic tree, X the set of leaves of T and Σ the set of labels of X. Then Σ is called the set of *taxa*.

Definition 2.19. Let T be a phylogenetic tree and X the set of leaves of T. Some $C \subset X$ is called a *clade* if $\exists\, v \in T$ s.t. C is the set of leaves in the induced subtree T_v of T. A clade C is called *trivial* if $|C| = 1$ or $C = X$. The set $\mathcal{C}(T) := \{C \subset V \mid C \text{ is a clade}\}$ is the set of clades of T, $\mathcal{C}^*(T) := \{C \in \mathcal{C}(T) \mid C \text{ is non trivial}\}$ the set of non trivial clades.

Definition 2.20. A rooted tree T is called *full binary*, if every node has either 0 or 2 children.

Remark.

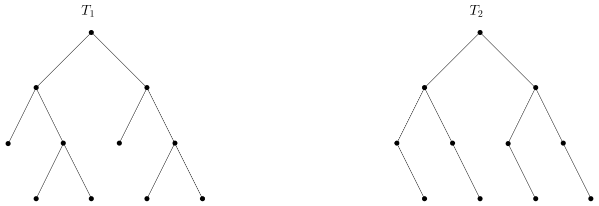


Figure 2.6: The illustration suggests, that the node 1 is the root of the tree. Together with some function $l: \{1, ..15\} \mapsto \{A, B, C\}$ T is a rooted labelled tree.

2.2 Other necessary Tools

Definition 2.21. Let $T = (V, E)$ be a rooted labelled tree. The so called *basic tree edit operations* on T are *relabelling, inserting* and *deleting*:

1. *Relabelling v:* changing the label of a node v.
2. *Inserting v underneath v′:* insert a new node v into T as a child of v' and assign the children of v' to the new node v. Denote the new tree by T', then:
$$C_{T'}(v') = \{v\} \text{ and } C_{T'}(v) = C_T(v')$$
.

3. *Deleting v underneath v':* the opposite transformation of inserting. Delete v, assign all children of v to v' in the same order. Denote the new tree by T', then:

$$C_{T'}(v') = C_T(v') \setminus \{v\} \cup C_T(v)$$

.

Definition 2.22. Let $T = (V, E)$ be a rooted labelled tree and let o be one of the basic edit operations defined above. We denote by $o(T)$ the tree that we get after executing the operation o on the tree T.
Furthermore let $o' = (o'_1, o'_2, ..., o'_n)$ be a finite sequence of basic edit operations. We define $o'(T)$ as the consecutive application of the basic operations:

$$o'(T) := o'_n(o'_{n-1}(...(o'_1(T)...)).$$

Definition 2.23. Let $T = (V, E)$ be a rooted labelled tree, let Σ be the set of labels and $\sigma, \sigma' \in \Sigma$. Furthermore let o be one of the basic edit operations defined above. Then the cost of $c(o)$ is defined as:

$$c(o) := \begin{cases} c_{rel}(\sigma, \sigma') & \text{Relabelling existing node } v \text{ from } \sigma \text{ to } \sigma' \\ c_{ins}(\sigma) & \text{Inserting new node } v \text{ with label } \sigma \\ c_{del}(\sigma) & \text{Deleting existing node } v \text{ with label } \sigma \end{cases}$$

Moreover let $o' = (o'_1, ...o'_n)$ be a finite sequence of basic edit operations. We

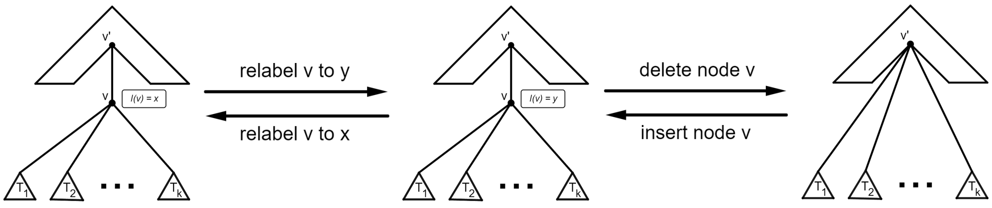


Figure 2.7: Illustration of the basic tree edit operations relabelling, deleting and inserting.

define the costs of o' to be the sum of costs of the individual operations:

$$c(o') := \sum_{i=1}^{n} c(o'_i).$$

Remark. Because of the symmetry we can assume $c_{del}(\sigma) = c_{ins}(\sigma)$. Because of that, we will only work with relabelling and deleting operations later on.

Definition 2.24. Let $T = (V, E)$ be a rooted labelled tree and let o be one of the basic edit operations defined above. Then the cost of $c(o)$

Definition 2.25. Let $T = (V, E)$ be a rooted labelled tree. A function $d : V \times V \mapsto \mathcal{R}$ is a *metric* if the following conditions are fulfilled $\forall u, v, w, \in V$:

1. $d(v, w) \geq 0$
2. $d(v, w) = 0 \iff v = w$
3. $d(v, w) = d(w, v)$
4. $d(u, w) \leq d(u, v) + d(v, w)$

Definition 2.26. The *Catalan numbers* $(C_n)_{n \in \mathbb{Z}_{\geq 0}}$ forms a sequence of natural numbers which occur in many recursively defined combinatorial problems. They are recursively defined as follows:

$$C_0 = 1C_n \qquad = \sum_{i}^{n-1} C_i C_{n-i-1} \text{ for } n \geq 1$$

Remark. The following two combinatorial problems are examples in which the Catalan numbers occur:

- Counting the number of expressions containing n pairs of parentheses where any prefix of the expression contains at least as many opening parentheses "(" as closing ones ")".
- Counting the number of full binary trees with $n + 1$ leaves.

3 Tree Edit Distance

In this chapter we will discuss the so called *tree edit distance*. This chapter is based on Demaine et al.'s paper about an optimal decomposition algorithm [].

3.1 Introduction

Definition 3.1. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two rooted ordered labelled trees together with a set of labels Σ. Furthermore let the costs for the basic edit operations relabelling, inserting and deleting be fixed.
Let $o^* := (o_1^*, ..., o_n^*)$ be a finite sequence of edit operations that fulfills the following:

$$o^*(T_1) = o_n^*(...(o_1^*(T_1)) = T_2$$

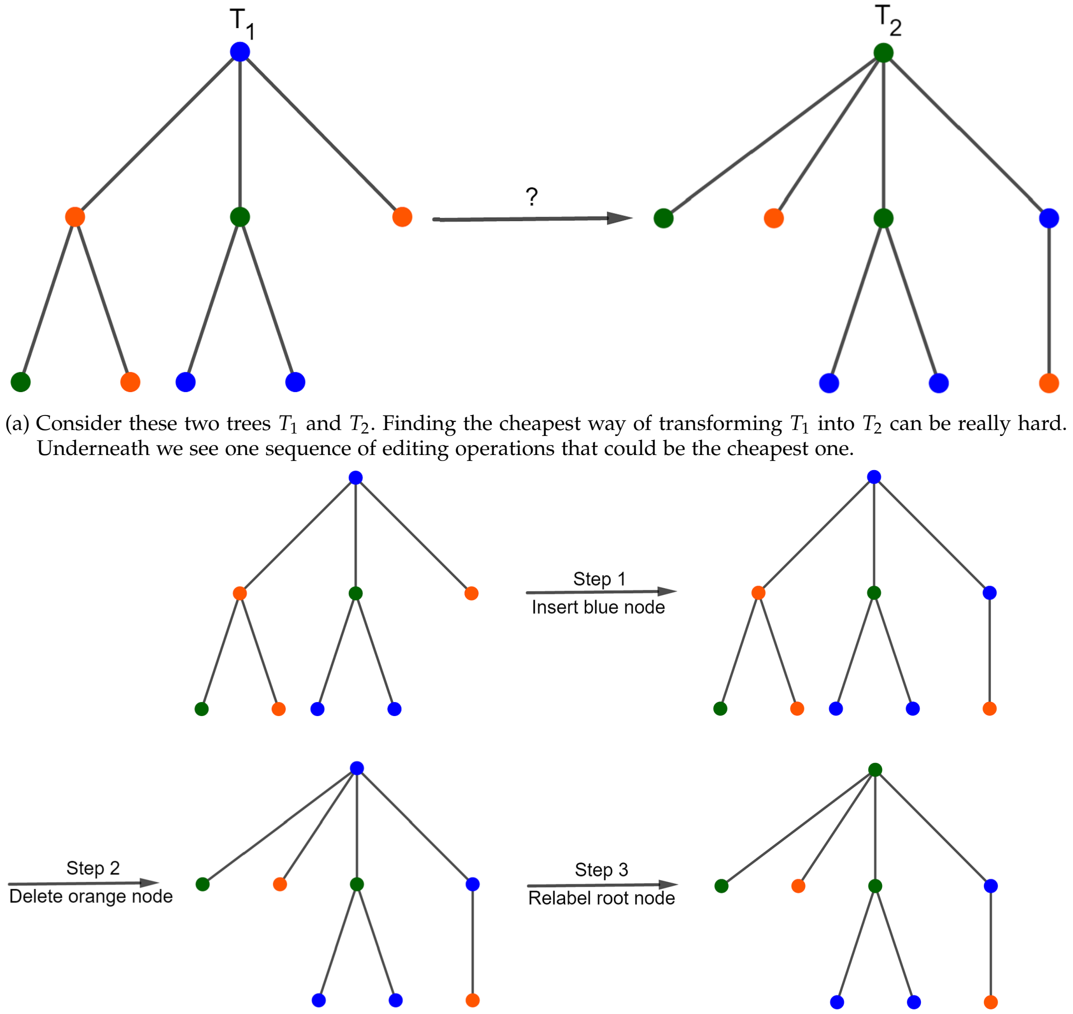
$$c(o^*) = \sum_{i=1}^{n} c(o_i^*) = \min_{\substack{o \text{ sequence of} \\ \text{edit operations}}} \{c(o) \,|\, o(T_1) = T_2\}$$

Then the *tree edit distance* $\delta(T_1, T_2)$ is defined as:

$$\delta(T_1, T_2) := c(o^*).$$

Definition 3.2. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two rooted ordered labelled forests and the rest as in the definition 3.1. Then we can extend the definition of the tree edit distance $\delta(T_1, T_2)$ trivially.

The origin of finding the tree edit distance is an intuitive way of comparing two trees. Given two labelled ordered trees T_1 and T_2. What is the cheapest sequence of edit operations that transforms T_1 into T_2? Take a look at the trees T_1 and T_2 in **??** and the sequence of operations that performs the transformation from T_1 to T_2. If we assume, for example, that every operation costs the same then the sequence shown in the figure would be an optimal one. The tree edit distance is used in the fields of structured text databases, computer vision and in bioinformatics. In the last field one



(a) Consider these two trees T_1 and T_2. Finding the cheapest way of transforming T_1 into T_2 can be really hard. Underneath we see one sequence of editing operations that could be the cheapest one.

(b) Step 1: Insert a node right above the rightmost child of the root. Step 2: Delete the leftmost child of the root. Step 3: Relabel the root node.

needs to compare the secondary structure of RNA-molecules without the disadvantages of other approaches. For a more detailed description about this topic please take a look at [].

3.2 Short History of the Tree Edit Distance

The tree edit distance was introduced by Tai in the year 1979 [15]. In addition to the definition he also provided an algorithm to compute the tree edit distance. The running time and space complexity amounts to $O(t_{l,1}^2 t_{l,2}^2 t_1 t_2)$ which implies a worst-case running time of $O(t_1^6)$.
It took about a century until Shasha and Zhang [14] came up with a dynamic programming approach that improved the running time to $O(t_1^2 t_2^2)$. Later on Klein [11] started with his algorithm, changed the way to of branching and was able to further improve the running time to $O(t_1 t_2^2 \log t_1)$ time. Dulucq and Touzet [7] proved a lower bound of $\Omega(t_1 t_2 \log t_1 \log t_2)$ on the running time for all algorithms for the tree edit distance which are based on dynamic programming in 2003. Finally, in 2007 Demaine et al. [6] provided an algorithm that satisfies the lower bound on dynamic programming approaches.
Chen [5] presented a different approach relying on fast matrix multiplication solving the tree edit distance problem in $O(t_1 t_2 + t_1 t_{2,l}^2 + t_{1,l} t_{2,l}^{2.5})$ time and $O(t_1 + (t_2 + t_{2,l}^2) \min\{t_{1,l}, t_{2,l}\})$ space. You can also take a look at other algorithms, see [1, 2, 16]

In this thesis we will concentrate on the dynamic programming approaches of Shasha and Zhang, Klein and last but not least Demaine et al. But since this thesis tries to show different comparing mechanisms we will keep it rather short. If you want to have a more detailed description about something, please take a look into the original papers.

3.3 Dynamic Programming Approach

The key for any dynamic programming approach is to find a suitable way for branching a hard problem into smaller and therefore easier subproblems.

Definition 3.3. Let T_1, T_2 be two rooted labelled ordered forests and let the cost functions be defined as stated in the definition. Consider the problem of computing $\delta(T_1, T_2)$ with a fixed dynamic programming approach A. A *relevant subproblem* of (T_1, T_2) is any pair of rooted labelled forests (T_1', T_2') such that the following holds:
During the computation of $\delta(T_1, T_2)$ we encounter the problem of solving $\delta(T_1', T_2')$. We define $\mathcal{R}_A$ to be the set of all relevant subproblems
We call (T_1', T_2') to be a trivial subproblem if and only if (T_1', T_2') is a relevant subproblem and $\exists i \in \{1,2\}$ s.t. $T_i' = (\emptyset, \emptyset)$. We defnine $\mathcal{T}_A \subset \mathcal{R}_A$ to be set of all trivial subproblems.

Remark. Since a relevant subproblem (T_1', T_2') occurs in the computation of $\delta(T_1, T_2)$, there has to exist a sequence of basic editing operations o only consisting of relabelling and deleting operations s.t. $o(T_1) = T_i' \, \forall i \in \{1,2\}$.

Remark. We only consider deleting and relabelling operations because of the symmetry between deleting a node in T_1 and inserting a proper node in T_2 vice versa.

Remark. A trivial dynamic programming approach would lead to $\Omega(2^{t_1+t_2})$ subproblems. Therefore it is necessary to branch in a smart way to get a polynomial number of relevant subproblems.

The idea of branching is to consider the two rightmost or leftmost roots of T_1 and T_2. Either one of those two gets deleted or the two roots are matched. In the first case you result in a new smaller relevant subproblem, in the latter you even end up with two separated relevant subproblems:
Suppose you match r_{T_1} with r_{T_2}. Then any node from R_{T_1} that gets matched with a node in T_2 has to be matched with a node in R_{T_2} because of the strict ancestry relation. So matching those two roots splits the problem of

computing $\delta(T_1, T_2)$ into the two subproblems of computing $\delta(R^\circ_{T_1}, R^\circ_{T_2})$ and $\delta(T_1 - R_{T_1}, T_2 - R_{T_2})$.

Definition 3.4. Consider the problem of computing the tree edit distance $\delta(T_1, T_2)$ with a dynamic programming approach A.
We call $S_A : \mathcal{R}_A \mapsto$ {left,right} the *decomposition strategy* of the dynamic programming approach A if for any relevant subproblem $(T_1', T_2') \in \mathcal{R}_A$ the following holds:
The direction $S_A((T_1', T_2'))$ coincides with the direction of branching according to the dynamic programming approach A.

3.3.1 Shasha and Zhang's algorithm

Shasha and Zhangs algorithm is the most basic dynamic programming approach. They restrict themselves to the decomposition strategy that always chooses the right direction.

Lemma 3.5. *Let T_1, T_2 be two rooted labelled forests and assume these forests are ordered according to the post order indexing. Consider the problem of computing $\delta(T_1, T_2)$ with a dynamic programming approach A that has a decomposition strategy $S_A(T_1', T_2') = right \, \forall (T_1', T_2') \in \mathcal{R}_A \setminus \mathcal{T}_A$. Then:*
$\forall (T_1', T_2') \in \mathcal{R}_A \setminus \mathcal{T}_A : \exists \, i_1 \leq j_1, \, i_2 \leq j_2 \in \mathcal{N}$ s.t.:

$$
\begin{aligned}
V(T_1') &= \{i_1, i_1 + 1, ..., j_2\} \\
V(T_2') &= \{i_2, i_2 + 1, ..., j_2\}.
\end{aligned}
$$

Remark (Remark 1)*.* Because of the post order indexing, the rightmost root r_T has the highest overall index.

Remark (Remark 2)*.* For two induced subtrees $T_{v'}$ and $T_{v''}$ of T s.t. $V(T') \cup V(T'') = \emptyset$ assume that v' lies on the left of v''. Then the index of every node in $T_{v'}$ is strictly smaller than the index of all nodes in $T_{v''}$.

Proof. We make an inductive argument. We start with our base: For T_1 and T_2 the claim is trivially true. So let's consider an induction step and assume that the claim holds for a relevant subproblem (T_1', T_2'). Therefore there exists the indexes i_1, j_1, i_2, j_2 as in the lemma. We have three possible induction steps:

1. *Delete $r_{T_1'}$:* If $i_1 \leq j_1 - 1$ the new indexes will be $i_1, j_1 - 1, i_2, j_2$ because of Remark 1. Otherwise we just deleted the only node left in $T_1' \Rightarrow$ the new relevant subproblem $(T_1'', T_2'') \in \mathcal{T}_A$
2. *Delete $r_{T_2'}$:* Equivalent to the previous case.
3. *Match $r_{T_1'}$ and $r_{T_2'}$:* As written previously we split the problem (T_1', T_2') into two subproblems $\delta(R^\circ_{T_1}, R^\circ_{T_2})$ and $\delta(T_1 - R_{T_1}, T_2 - R_{T_2})$. Assume both subproblems are not trivial. Using Remark 2 we see that all nodes in R_{T_1} have a higher index than all the nodes in $T_1 - R_{T_1}$.
   $\Rightarrow \exists k_1$ s.t. $i_1 < k_1 < j_1$ with $V(T_1 - R_{T_1}) = \{i_1, ...k_1 - 1\}$ and $V(R_{T_1}) = \{k_1, ..., j_1\}$ and k_2 equivalently, closing our induction step argument.

□

Lemma 3.5 provides us with a trivial upper bound of relevant subproblems of $O(t_1^2 t_2^2)$ since there are only $\binom{t_1}{2} = O(t_1^2)$ such sets for T_1 and $\binom{t_2}{2} = O(t_2^2)$ such sets for T_2 respectively.

Lemma 3.6. *Let T_1, T_2 be two rooted ordered labelled forests with the set of labels Σ. Assume that the cost functions for the basic tree edit operations are fixed. Then one can compute $\delta(T_1, T_2)$ considering $O(\min\{t_{1,l}, t_{1,h}\} \min\{t_{2,l}, t_{2,h}\} t_1 t_2)$ subproblems with the following recursion steps:*

1. $\delta(\emptyset, \emptyset) = 0$;
2. $\delta(T_1, \emptyset) = \delta(T_1 - r_{T_1}, \emptyset) + c_{del}(r_{T_1})$;
3. $\delta(\emptyset, T_2) = \delta(\emptyset, T_2 - r_{T_2}) + c_{del}(r_{T_2})$;

4.

$$\delta(T_1, T_2) = \min \begin{cases} \delta(T_1 - r_{T_1}, T_2) + c_{del}(r_{T_1}) \\ \delta(T_1, T_2 - r_{T_2}) + c_{del}(r_{T_2}) \\ \delta(R^\circ_{T_1}, R^\circ_{T_2}) + \delta(T_1 - R_{T_1}, T_2 - R_{T_2} + c_{rel}(r_{T_1}, r_{T_2}) \end{cases}$$

Proof. Correctness: Constraint 1 is trivial. Constraints 2 and 3 handle the case of trivial subproblems: We just delete all nodes and add the costs for doing so. For a non-trivial relevant subproblem we have to find the cheapest way of procedure: Either delete a rightmost root or match them. The equations are trivial.

Running time: We calculate an upper bound on the number of different subforests of T_1 and T_2 that appear in any relevant subproblem independently and multiply those bounds together. Thus we get an overall upper bound on the number of relevant subproblems.
For that we have to take a closer look at the role of keyroots and prefixes as defined in the first chapter: Suppose T_1' is a subforest of T_1 that appears in some subproblem $\Rightarrow \exists i_{T_1'}, j_{T_1'}$ s.t. $V(T_1') = \{i_{T_1'}, ..., j_{T_1'}\}$.
If $i_{T_1'} = 1$ then, under the assumption that $j_{T_1'} < t_1$, T_1' is a prefix of $T_1^\circ = (T_1)^\circ_{r_1}$ where r_1 is the root of T_1.
If $i_{T_1'} > 1$ then there has to exist an induced subtree that lies completely on the left of T_1', even if it is only the leftmost leaf. Therefore there exists a biggest subtree that lies completely on the left of T_1'. This subtree will be of the form $(T_1)_v$ for some $v \in V(T_1)$. Thus T_1' has to be a prefix of the right sibling w of v.
In both cases we end up with the statement, that any subforest of T_1, appearing in a relevant subproblem, is a prefix of an induced subtree $(T_1)_v$ for some $v \in$ keyroots(T_1). This implies summing up all such prefixes will be an upper bound on the number of relevant subproblems:

$$\sum_{v \in \text{keyroots}(T_1)} |(T_1)^\circ_v| = \sum_{v \in T_1} \text{cdepth}(v) \le \sum_{v \in T_1} \text{cdepth}(T_1) = |T_1|\text{cdepth}(T_1)$$

Shasha and Zhang went on to prove the missing inequality:

$$\text{cdepth}(T_1) \leq \min\{t_{1,l}, t_{1,h}\}$$

Combining all the parts together we reach target running time of $O(\min\{t_{1,l}, t_{1,h}\} \min\{t_{2,l}$

□

3.3.2 Klein's algorithm

Klein [11] improved the algorithm of Shasha and Zhang by using a more advanced decomposition strategy. The idea is to compare the sizes of the two outermost trees of T_1:

Definition 3.7. Let (T'_1, T'_2) be any relevant subproblem for computing $\delta(T_1, T_2)$. Klein's decomposition strategy S_K is defined as follows:

$$S_K(T'_1, T'_2) = \begin{cases} \text{left} & |V(L_{T'_1})| \leq |V(R_{T'_1})| \\ \text{right} & \text{otherwise} \end{cases}$$

Klein's algorithm improved the running time of decomposition algorithms to $O(n^3 \log n)$. The proof of this running time makes use of the heavy path decomposition and an upper bound of $\log(T) + O(1)$ on the ldepth(v). The key idea is that every relevant subproblem can be obtained by some $i < |T_v|$ consecutive deletions from T_v for some light node $v \in V(T)$.

3.3.3 Demaine et al.'s optimal Algorithm

Demaine et al. presented a new algorithm based on the dynamic programming approach. They proved a running time of $O(t_2^2 t_1(1 + \log \frac{t_1}{t_2}))$ and lastly showed that this satisfies a lower bound on the running time of dynamic programming approaches for the tree edit distance.

This subsection contains statements without proofs for the sake of this thesis

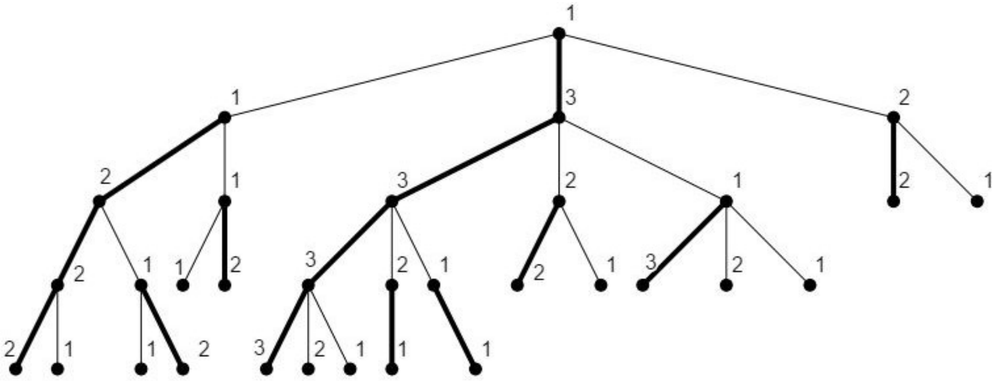


Figure 3.2: The number at each node indicates the order among siblings in which they are considered according to S_D.

length. If you are interested in them, we would like to refer to the original paper of Demaine et al. [6]

Definition 3.8. Let (T_1', T_2') be any relevant subproblem for computing $\delta(T_1, T_2)$. Demaine et al.'s decomposition strategy S_D is defined as follows:

$$S_D(T_1', T_2') = \begin{cases} \text{left} & \text{if } T_1' \text{ is a tree or if } l_{T_1'} \text{ is not the heavy child of its parent} \\ \text{right} & \text{otherwise} \end{cases}$$

Remark. Take a look at Figure **??**. As the caption explains the number at each node shall demonstrate the order among children. If you take a look at the children of the root, for example, the first direction according to S_D would be left, the second one would be right and last but not least the heavy child of the root would be considered.
It is trivial, that according to S_D the heavy child of a node will always be considered at last.

Lemma 3.9. *Let T_1, T_2 rooted ordered labelled trees be given. Suppose we want to compute $\delta((T_1)_v, T_2)$ with $v \in TopLight(T_1)$. Then we encounter all pairs $((T_1)_u^\circ, (T_2)_w^\circ)$ where $u \in T_1, w \in T_2$ and both not on the main heavy paths of T_1, T_2 respectively as relevant suproblems and therefore compute $\delta((T_1)_u^\circ, (T_2)_w^\circ)$.*

Combining this lemma and the decomposition strategy S_D leads to the following algorithm:

Theorem 3.10. *We compute $\delta(T_1, T_2)$ recursively as follows:*

1. *If $|V(T_1)| < |T_2|$ compute $\delta(T_2, T_1)$ instead.*
2. *Recursively compute $\delta((T_1)_v, T_2)\ \forall v \in TopLight(T_1)$ using these recursive steps.*
3. *Compute $\delta(T_1, T_2)$ using the decomposition strategy S_D. However do not recurse into subproblems that have previously been computed in step 2.*

Using these steps, we can compute $\delta(T_1, T_2)$ in $O(t_2^2 t_1(1 + \log \frac{t_1}{t_2}))$ time.

3.3.4 Lower bound on Decomposition Algorithms

The potential of decomposition algorithms has a bound that has already been achieved by the previous algorithm of Demaine et al. To finish this chapter on the tree edit distance we will present a proof of a lower bound of $\Omega(t_2^2 t_1)$ and illustrate the structure of trees which are used to show the actual lower bound of $\Omega(t_2^2 t_1(1 + \log \frac{t_1}{t_2}))$.

Lemma 3.11. *For any decomposition algorithm solving the tree edit distance problem, there exists a pair of trees (T_1, T_2) with sizes t_1, t_2 resprectively, such that the number of relevant subproblems is $\Omega(t_2^2 t_1)$*

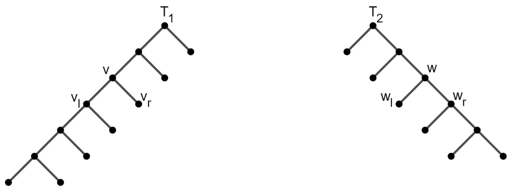


Figure 3.3: Sketch of T_1 and T_2 which fulfill a lower bound on the running time for each decomposition algorithm of $\Omega(t_2^2 t_1)$

Proof. Let S be the strategy of decomposition algorithm and assume T_1 and T_2 to be as in Figure 3.3. As previously stated, every pair $((T_1)_v^\circ, (T_2)_w^\circ)$ for

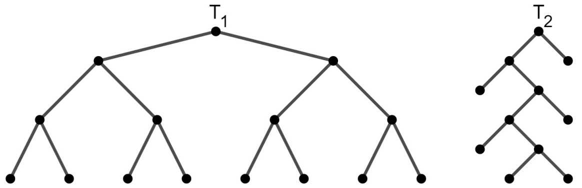


Figure 3.4: T_1 and T_2 used to prove a lower bound of $\Omega(t_2^2 t_1(1+\log\frac{t_1}{t_2}))$

$v \in T_1, w \in T_2$ is a relevant subproblem for S. We count the number of such relevant subproblems, where v and w are inner nodes of T_1 and T_2. For an inner node x, let us denote the left child of x as x_l and the right child of x as x_r. In the forest $(T_1)^\circ_v$ the rightmost root is v_r and in $(T_2)^\circ_w$ the leftmost root is w_l. In each step S decides the direction from which side we should delete. Every time the strategy chooses left, we delete from T_1 and otherwise from T_2. 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 $\min\{|(T_1)^\circ_v|, |(T_2)^\circ_w|\}$ steps until every relevant subproblem of $((T_1)^\circ_v, (T_2)^\circ_w)$ is found. Since v_r and w_l are the outermost roots, the computational paths of $((T_1)^\circ_v, (T_2)^\circ_w)$ and $((T_1)^\circ_{v'}, (T_2)^\circ_{w'})$ are completely disjoint. Because of their structure there are $\frac{t_1}{2}$ and $\frac{t_1}{2}$ internal nodes in T_1 and T_2 respectively, yielding the following equation:

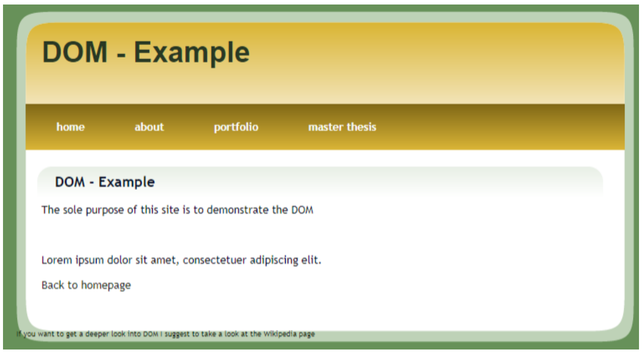
$$\sum_{(v,w)\text{internal nodes}} \min\{|(T_1)^\circ_v|, |(T_2)^\circ_w|\} = \sum_{i=1}^{\frac{t_1}{2}} \sum_{j=1}^{\frac{t_2}{2}} \min\{2i, 2j\} = \Omega(t_2^2 t_1).$$

□

In the case of $t_2 \neq \Theta(t_1)$ this bound doesn't match the running time of Demaine et al.'s algorithm. But considering trees structured as the ones in Figure 3.4, one can proof the actual lower bound of $\Omega(t_2^2 t_1(1+\log\frac{t_1}{t_2}))$

4 Flexible Tree Matching

One problem of the standard tree edit distance is the strict requirement regarding the tree parent-to-child relationship within an ordered tree, the so called hierarchy, and ordering among siblings. If a node gets mapped while computing the tree edit distance, its children have to get mapped to some descendants of this mapped node. In some domains, the most appropriate matching may not follow these requirements. One of these domains is the DOM (Document Object Model) of a website. A standard HTML-based website can easily be structured according to the respective tags. Take a look at the website in Figure 4.1 and the its HTML-code in Listing 4.1.



DOM - Example

The sole purpose of this site is to demonstrate the DOM

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Back to homepage

If you want to get a deeper look into DOM I suggest to take a look at the Wikipedia page

Figure 4.1: The example website.

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" type="text/css" href="style.css">
    <title>DOM-example by Clemens Andritsch</title>
  </head>
```

```
 <body>
  <div id="page">
   <div id="header">
    <div id="headerTitle">DOM - Example</div>
   </div>
   <div id="bar">
    <a href="#">home</a>
    <a href="#">about</a>
    <a href="#">portfolio</a>
    <a href="#">master thesis</a>
   </div>
   <div id="cont1" class="contentTitle">
   <h1>DOM - Example</h1>
   </div>
   <div id="cont2" class="contentText">
    <p id="p1">The sole purpose of this site is to demonstrate the DOM</p>
    <p id="p2">&nbsp;</p>
    <p id="p3">Lorem ipsum dolor sit amet, consectetuer adipiscing elit.</p>
    <p id="p4">
     <a href="index.html">Back to homepage</a>
    </p>
   </div>
  </div>
  <div id="footer">
   <p id="footer-text">If you want to get a deeper look into DOM I suggest to take a look at the <a href="https://de.wikipedia.org/wiki/Document_Object_Model">Wikipedia page</a></p>
  </div>
 </body>
</html>
```

Listing 4.1: Html code of DOM example

The DOM-tree of the example website is intuitively clear: The outermost tag, the <html>-tag, is the tree's root. The root's children are the <head>-tag and the <body>-tag and so on. This leads to complete DOM-tree illustrated in Figure 4.2.

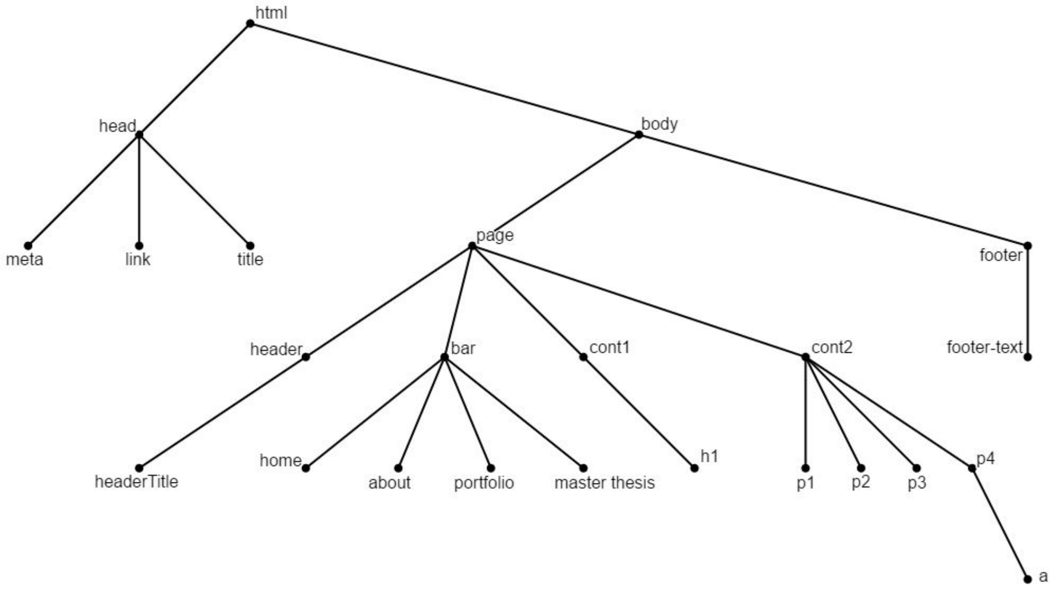
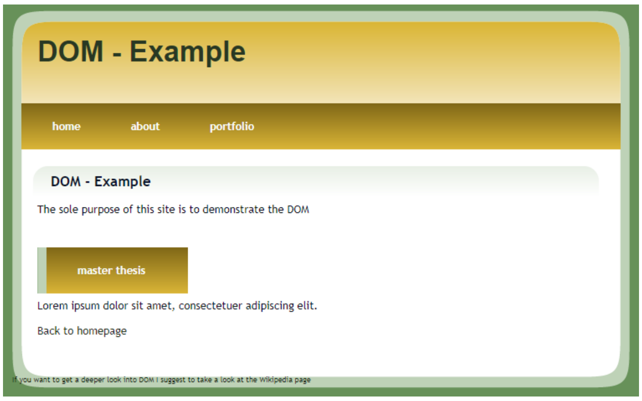


Figure 4.2: Complete DOM-tree of the example website.

Intuitively, any small change to the website should not lead to a big distance between the two DOM-trees. Suppose one changes the order of the buttons in the header menu as well as moving one of these buttons into the content area of the website as seen in Figure 4.3. The website and its functionalities remain quite similar, but it would take about three deletions and four insertions to end up with the new DOM-tree. Obviously it would be cheaper to delete the whole header menu and therefore reduce the functionality of the website.

This kind of issue arises frequently in the context of comparing tree models. Flexible tree matching models have been introduced in an effort to appropriately handle the above mentioned issues. Instead of requiring strict left-to-right ordering and hierarchy conditions, one may relax them by introducing costs to penalize the violation of this type of requirements. Kumar et al. [12] developed an algorithm that matches nodes with similar labels and penalizes edges that break up sibling groups or violate the hierachy. In the example above, moving the button from header menu into the content is such a violation of the hierarchy, because a node gets shifted



DOM - Example

DOM - Example

The sole purpose of this site is to demonstrate the DOM

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Back to homepage

If you want to get a deeper look into DOM I suggest to take a look at the Wikipedia page

Figure 4.3: Small changes to the website should not affect the distance of the respective DOM-trees.

into another subtree. This needs to have some kind of costs associated to it, but it should definitely be cheaper than deleting the subtree and inserting it again in some other place.

Finding the minimum flexible tree edit distance can be reduced to finding a minimum cost matching in a flexible cost model. Kumar et al. showed that finding the flexible tree edit distance is strongly $\mathcal{NP}$-complete. This implies that there are no efficient algorithms to compute the minimum flexible tree edit distance. However, there are some approximating heuristics.

This chapter is based on the previously cited paper by Kumar et al.

4.1 The Model for the Flexible Tree Edit Distance

Let T_1 and T_2 be the two trees for which the flexible tree edit distances shall be computed. We construct a complete bipartite graph G with $G := (\{V(T_1) \cup \otimes_1\} \dot{\cup} \{V(T_2) \cup \otimes_2\}, E)$. Hence the edge set E is defined as the set $E := \{\{v_1, v_2\} \mid v_i \in \{V(T_i) \cup \otimes_i\} \text{ for } i = 1, 2\}$. The nodes $\otimes_i$ are so called *no-match* nodes. Every edge $e = \{v_1, v_2\} \in E$ represents matching a node v_1 to a node v_2. If $v_2 = \otimes_2$, the edge e would represent the deletion of v_1, since v_1 was not matched to any node from the tree T_2. An analogous statement holds for an edge $\{\otimes_1, v_2\}$. Every edge $e \in T_1 \times T_2$ is assigned some cost function $c(e) = c_r(e) + c_a(e) + c_s(e)$. The exact definitions follow later. In short terms, these three summands represent the costs that we mentioned earlier in this chapter: c_r represents the costs of relabelling a node, c_a penalizes violations of ancestry relationships and c_s punishes broken up sibling groups. Edges connecting tree nodes with no-match nodes have a fixed constant cost w_n, only depending on the number of nodes in the trees. The goal is to find a minimum cost flexible matching $M \subset E$ such that every node of $T_1 \cup T_2$ is covered exactly once while the no-match nodes may be covered multiple times. We call the set of all such flexible matching M_{T_1,T_2}. Suppose that $v \in V(T_1)$ and $w \in V(T_2)$ and let $e := \{v, w\} \in E$. The costs for relabelling e, i.e. $c_r(e)$, only depend on the nodes v and w themselves. They are fixed for every edge and are known before starting to match nodes. $c_a(e)$ and $c_s(e)$ on the other hand may depend on the choice of the flexible matching M. To be more precise the costs $c_a(\{v, w\})$ are linearly dependent on the number of children of v that do not get mapped onto children of w. Define $M(v) \in \{T_2 \cup \otimes_2\}$ to be the node that v is mapped onto according to the flexible matching M and suppose that $M(v) \neq \otimes_2$. Moreover, define $C(v) \subset T_1$ to be the set of children of node v. Then we can define $V(v)$ to be the set of children of v that violate the ancestry condition, i.e.:

$$V(v) := \{v' \in C(v) \mid M(v') \in T_2 \setminus C(M(v))\}$$

As stated previously, the cost function $c_a(e)$ is linearly dependent on the sizes of the sets $V(v)$ and $V(w)$ with some constant factor ω_a:

$$c_a(\{v, w\}) := \omega_a(|V(v)| + |V(w)|)$$

We need further tree related concepts before we can define the cost function $c_s(e)$ in a straight forward way. Therefore $P(v) \in T_1$ is defined as the parent of the node v and $S(v) := C(P(v))$ is the sibling group of v. If v is the root of the tree, then $P(v)$ does not exist and $S(v) := \{v\}$. Note that the sibling group $S(v)$ always contains the node v itself and thus is not empty. For a matching M, we define the sibling-invariant subset of v, $I_M(v)$, to be the siblings of v which are mapped into the same sibling group as v:

$$I_M(v) = \{v' \in S(v) \mid M(v') \in S(M(v))\}$$

Accordingly the sibling-divergent subset of v, $D_M(v)$, are the siblings of v which are mapped to a node in $T_2 \setminus S(M(v))$:

$$\begin{aligned} D_M(v) :&= \{v' \in S(v) \mid M(v') \in T_2 \setminus S(M(v))\} \\ &= \{v' \in S(v) \setminus I_M(v) \mid M(v') \neq \otimes_2\} \end{aligned}$$

Finally, we define the set of distinct sibling families to be the set of all sibling groups, that the siblings of v map into:

$$F_M(v) = \bigcup_{v' \in S(v)} P(M(v'))$$

Now we can define the costs for sibling group violations depending on a constant ω_s:

$$c_s(\{v,w\}, M) := \omega_s(\frac{|D_M(v)|}{|I_M(v)||F_M(v)|} + \frac{|D_M(w)|}{|I_M(w)||F_M(w)|})$$

One can show, that the costs $c_s(\{v,w\}, M)$ increase, if a node in the sibling group of v or w gets reassigned to some node outside of the corresponding sibling group.

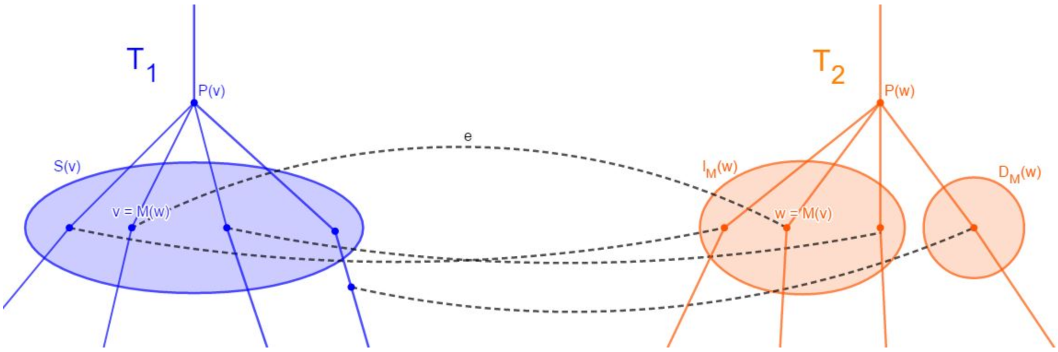


Figure 4.4: A visual representation of the above described tree concepts.

Definition 4.1. Let T_1, T_2 be two rooted ordered trees. Let $G := (\{V(T_1) \cup \otimes_1\} \dot\cup \{V(T_2) \cup \otimes_2\}, E)$ be a graph as defined above. Furthermore let constants ω_n, ω_a, ω_s and the relabelling function $c_r(e)$ be given.
Let $M^* \in M_{T_1,T_2}$ be a flexible matching that covers each node $v \in T_i$, $i \in \{1,2\}$ exactly once and that fulfills the following equation:

$$c(M^*) := \sum_{e^* \in M^*} c(e^*) = \min_{M \in M_{T_1,T_2}} \sum_{e \in M} c(e)$$

Then we call $c(M^*)$ the *flexible tree edit distance*.

4.2 Approximation and Conclusion

As spoiled in the introduction of this section, computing the flexible tree edit distance is $\mathcal{NP}$-hard. There is a short and elegant proof, that presents a reduction of the 3-partition problem to the flexible tree matching problem. Once again you can find the details in the paper of Kumar et al [12]. Hence, there exists no efficient algorithm that computes the flexible tree matching to optimality. But there are stochastic optimization algorithms to get an approximation of the flexible tree edit distance. Kumar et al. [12] presented a Monte Carlo algorithm where they fix edges one after another, prune all other incident edges to the endpoints of the current edge and update the bounds for all other nodes. They start with an empty flexible matching M and calculate bounds for the values of $c_a(e)$ and $c_s(e)$ for all edges e. After including an edge $e_1 = \{v, w\}$ into M, they delete all other adjacent edges to v and w and update the bounds for the cost functions c_a and c_s for all remaining edges. Naturally the more edges are fixed, the better the bounds get. For a more detailed description of the actual algorithm, please take a look at the cited paper. The authors also described how to adapt the cost factors ω_r, ω_s, ω_a and ω_n step by step to improve the results.
Depending on the application, the flexible tree matching can have huge advantages with respect to the classic tree edit distance, especially in fields where hierarchy is suggestive rather than definitive. In different applications

sibling group violations or ancestry violations may have different significance and their significance can be modelled by appropriately chosen values for coefficients ω_r and ω_s. Hence the coefficients of the cost model for the flexible tree matching can be tuned so as to reflect the real life problem as accurately as possible. If you have a database of exemplar matchings you may even implement a learning cost model that improves the cost factors to follow your needs. Nevertheless, there can not be an efficient algorithm to calculate the flexible tree edit distance. So all results are more suggestive than definitive, just like the problem itself.

5 Robinson Foulds Metric

A field, where tree edit distances get applied a lot is the field of computational biology and bioinformatics. Comparing the structure of different RNAs is a classic example of such an application. An even more important one is comparing phylogenetic trees. A phylogenetic tree or evolutionary tree is a rooted branching diagram, that shows the evolutionary connectedness of different species. Multiple species can have recent common ancestor (in the evolutionary sense). This ancestor is represented as a node with an edge to all the descended species. A leaf is called a taxon and is labelled with the species it represents. All in all, one can build one huge phylogenetic tree that represents all life on earth. For the studies of evolutionary biology, scientists often need to compare different evolutionary theories regarding a certain subset of species. Therefore they have to take a look at the structure of the corresponding subtrees and apply some distance measurement on them.

5.1 Additional Background

The scientific field of phylogenetics is the field of evolutionary relationships and history among species and groups of organisms. This common ancestry is described as a phylogenetic tree. The taxons, the labels of the leaves of the phylogenetic tree, are hereby the species currently investigated. The interior nodes represent some common ancestor of the investigated species. If we

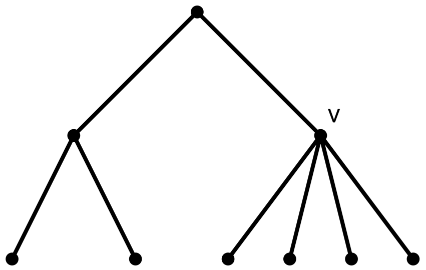


Figure 5.1: A rooted phylogenetic tree. The node v is a multifurcation since its degree is larger than 3. The ancestry relations aren't fully resolved for the children of v.

create a rooted phylogenetic tree, then the root would be the common ancestor of all these species. Sometimes the structure can't be fully resolved. This results in so called *multifurcations*, interior nodes of degree higher than three. Multifurcations may occur due to missing data for inferring the phylogeny. Often they appear in consensus trees, when partially contradictory trees were obtained by some methods. A perfectly resolved phylogenetic tree doesn't have any multifurcations implying a binary phylogenetic tree.

5.2 The original metric

The most commonly used distance measurement was introduced by Robinson and Foulds []. They introduced the term clade, which describes a group of leaves that have a common ancestor which they do not share with any other node.

The Robinson-Foulds metric is quite intuitive and easy to compute. It is the average number of non-trivial clades that are present in exactly one of the two trees:

Definition 5.1. Let T_1, T_2 be two trees with the same set of taxa X. Then we define the Robinson-Foulds metric d_{RF} as follows:

$$d_{RF} := \frac{1}{2}|\mathcal{C}^*(T_1) \triangle \mathcal{C}^*(T_2)|$$

Remark. Assuming that two trees T_1,T_2 have the same set of taxa X with $n := |X|$ implies that the number of interior nodes is bounded by $n-1$ for both trees. Furthermore, since we ignore trivial clades, the clade X, induced by the tree's root, will be ignored. Thus we end up with a maximum RF-distance of $n-2$ for two trees with the same taxa sets of size n.

Although the Robinson-Foulds metric is commonly used, it has some well known downsides. For example changing the position of a single leaf can yield a new tree having maximal distance from the original one. Let's take a look at Figure **??**. Let T_1 be the top left tree and T_2 be the top right one. It is easy to see that the set of clades are the following:

$$\begin{aligned}\mathcal{C}^*(T_1) &= \{\{1,...,j\} \mid j \in \{2,...,7\}\}\\ \mathcal{C}^*(T_2) &= \{\{2,...,j\} \mid j \in \{3,...,7\}\} \cup \{1,8\}\end{aligned}$$

Obviously in tree T_1 every clade contains both nodes 1 and 2. On the other hand in tree T_2 every clade either contains node 1 or 2, but never both of them. This implies that

$$\begin{aligned}\mathcal{C}^*(T_1) \cap \mathcal{C}^*(T_2) &= \emptyset\\ d_{RF} &= \frac{1}{2}|\mathcal{C}^*(T_1) \triangle \mathcal{C}^*(T_2)| = \frac{1}{2}(|\mathcal{C}^*(T_1)| + |\mathcal{C}^*(T_2)|)\\ d_{RF} &= \frac{1}{2}(6+6) = 6\end{aligned}$$

However the third tree T_3 also has the same RF distance of 6 to both trees T_1 and T_2. This points to another big disadvantage of the Robinson Foulds distance. The distribution of the distances is very much concentrated on the upper end of the scale. Two arbitrary phylogenetic trees with n leaves on the same set of taxa have a high probability to have a Robinson Foulds distance of $n-2$. Furthermore the example shows that the Robinson Foulds distance does not compare the structure of the two trees. Otherwise the distance between T_1 and T_2 would be much smaller than the one to the tree T_3.

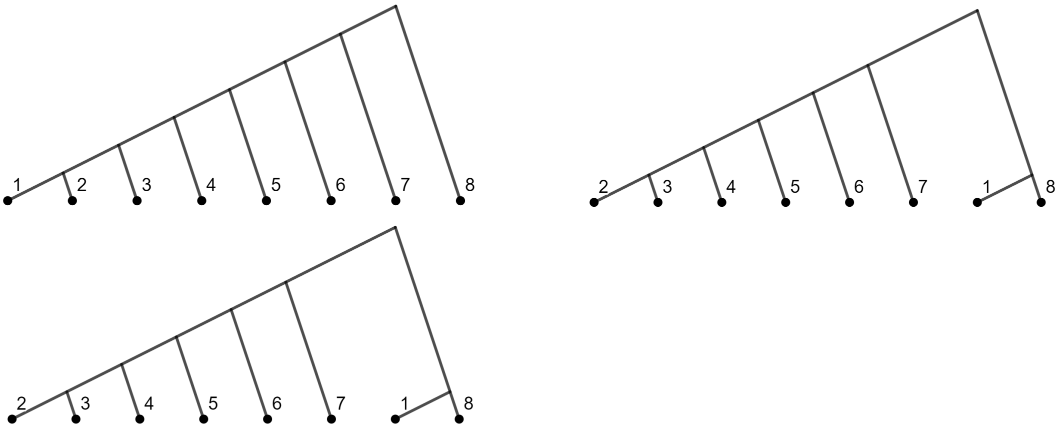


Figure 5.2: Three trees having the maximal RF-distance on the set of taxa $\{1, ..., 8\}$

5.3 The Generalized Robinson Foulds

To take advantage of structural similarities between T_1, T_2 Böcker et al. [3] suggested to extend the Robinson Foulds metric. They wanted to relax the condition of counting any clade which appears in the set of clades for one but not both trees. Therefore they introduced a bipartite graph $G(T_1, T_2)$ depending on the two trees under consideration.

Definition 5.2. Let T_1, T_2 be two phylogenetic trees with the same number of leaves n and on the same set of taxa X. We define the complete bipartite graph G_{T_1,T_2} on the following set of nodes:

$$G_{T_1,T_2} : \mathcal{C}^*(T_1) \times \mathcal{C}^*(T_2)$$

Definition 5.3. Let T_1, T_2 be two phylogenetic trees with the same number of leaves n and on the same set of taxa X. We introduce a cost function δ as follows:

$$\delta : \mathcal{P}(X) \times \mathcal{P}(X) \mapsto \mathcal{R}_{\geq 0} \cup \{\infty\}$$

where $\mathcal{P}(X)$ denotes the power set of X. The value $\delta(C, C')$ determines the dissimilarities between two clades C, $C' \subset X$. The value of $\delta(C, \emptyset) > 0$ denotes the dissimilarity bewtween a clade $C \in \mathcal{C}^*(T_1)$ with the empty clade in $\mathcal{C}^*(T_2)$. The value $\delta(\emptyset, C')$ is defined analogously.

Definition 5.4. Let T_1, T_2 be two phylogenetic trees with the same number of leaves n and on the same set of taxa X. Let G_{T_1,T_2} and a cost function δ be given as defined in Definitions 5.2 and 5.3. Furthermore let $M \subset \mathcal{C}^*(T_1) \times \mathcal{C}^*(T_2)$ be a matching in G_{T_1,T_2}. We call a clade $C \in \mathcal{C}^*(T_1)$ unmatched, if $(C, C') \notin M \; \forall C' \in \mathcal{C}^*(T_2)$, analogously for clades of T_2.
Now we can define the costs $d_\delta(M)$ of a matching M as:

$$d_\delta(M) := \sum_{(C,C') \in M} \delta(C, C') + \sum_{\substack{C \in \mathcal{C}^*(T_1), \\ C \text{ unmatched in } M}} \delta(C, \emptyset) + \sum_{\substack{C' \in \mathcal{C}^*(T_2), \\ C' \text{ unmatched in } M}} \delta(\emptyset, C')$$

Minimizing over all matchings in G_{T_1,T_2} yields $\overline{d}_\delta(T_1, T_2)$:

$$\overline{d}_\delta(T_1, T_2) := \min_{M \text{ matching in } G_{T_1,T_2}} d_\delta(M)$$

Lemma 5.5. *Let T_1, T_2 be two phylogenetic trees with the same number of leaves n and on the same set of taxa X. There is a distance function δ_{RF} s.t.*

$$\overline{d}_{\delta_{RF}}(T_1, T_2) = d_{RF}(T_1, T_2)$$

Proof. Let $\delta_{RF} : \mathcal{P}(X) \times \mathcal{P}(X) \mapsto \mathcal{R}_{\geq 0} \cup \{\infty\}$ be given as follows:

$$\delta_{RF}(C, C') = \begin{cases} 0 & \text{if } C = C' \\ \frac{1}{2} & \text{if } C = \emptyset \text{ or } C' = \emptyset \\ \infty & \text{if } C \neq C' \text{ and } C \neq \emptyset \neq C' \end{cases}$$

Let $M^* \subset \mathcal{C}^*(T_1) \times \mathcal{C}^*(T_2)$ be a matching s.t.:

$$\overline{d}_\delta(T_1, T_2) = \min_{M \text{ matching in } G_{T_1,T_2}} d_\delta(M) = d_\delta(M^*)$$

Since the empty matching has a finite value, M^* must not contain any edge (C, C') s.t. $\delta_{RF}(C, C') = \infty$. Therefore M^* only contains edges where the respective clades are congruent. Thus M^* minimizes the number of unmatched clades, counts them and divides this number by 2. Thus:

$$\overline{d}_{\delta_{RF}}(T_1,T_2) = \frac{1}{2}|\mathcal{C}^*(T_1) \triangle \mathcal{C}^*(T_2)| = d_{RF}(T_1,T_2)$$

□

Lemma 5.6. *Let T_1, T_2 be two phylogenetic trees with the same number of leaves n and on the same set of taxa X. Let G_{T_1,T_2} and a cost function δ be given as defined in Definitions 5.2 and 5.3. The task of computing the matching M^* that satisfies $\overline{d}_\delta(T_1,T_2) = d_\delta(M^*)$ can be simplified by the following model:*
For any edge $(C,C') \in E(G_{T_1,T_2})$ let its weight be given by

$$\omega(C,C') := \delta(C,\emptyset) + \delta(\emptyset,C') - \delta(C,C'). \tag{5.1}$$

Finding a minimal cost matching M of clades is now equivalent to finding a maximal cost matching M^* in G_{T_1,T_2}.
Furthermore, if δ is a metric, all weights are non-negative.

Proof. Let M^* be a maximal cost matching in G_{T_1,T_2}. Then the following implications are trivial:

$$\begin{aligned}
\sum_{\{C,C'\}\in M^*} \omega(C,C') &= \max_M \sum_{\{C,C'\}\in M} \delta(C,\emptyset)+\delta(\emptyset,C')-\delta(C,C')\\
&= \max_M \underbrace{\sum_{C\in\mathcal{C}^*(T_1)} \delta(C,\emptyset)}_{const.\ K_1} - \sum_{\substack{C\in\mathcal{C}^*(T_1),\\ C\text{ unmatched}}} \delta(C,\emptyset) + \underbrace{\sum_{C'\in\mathcal{C}^*(T_2)} \delta(\emptyset,C')}_{const.\ K_2}\\
&\qquad - \sum_{\substack{C\in\mathcal{C}^*(T_1),\\ C\text{ unmatched}}} \delta(\emptyset,C') - \sum_{\{C,C'\}\in M} \delta(C,C')\\
&= K_1+K_2+\max_M -\left(\sum_{\substack{C_1\in\mathcal{C}^*(T_1),\\ C\text{ unmatched}}} \delta(C,\emptyset) + \sum_{\substack{C\in\mathcal{C}^*(T_1),\\ C\text{ unmatched}}} \delta(\emptyset,C') + \sum_{\{C,C'\}\in M} \delta(C,C')\right)\\
&= K_1+K_2-\min_M d(M) = K_1+K_2-d(M^*)
\end{aligned}$$

Additionally, if δ is a metric, the triangle inequality holds true. Thus all all weights are non-negative. □

Let's compute the distances defined in Definition 5.3of the trees T_2, T_2 from Figure **??** with respect to the costs of the symmetric difference between sets:

$$\delta_{sym}(C, C') = |C \triangle C'| = |C \cup C'| - |C \cap C'|$$

In this case, the best way to match a clade $\{1, ..., j\} \in \mathcal{C}^*(T_1), j \geq 3$ to a clade in $\mathcal{C}^*(T_2)$ is to match it with the clade $\{2, ..., j\}$, since its symmetric difference is only 1. This handles all cases except for the clade $\{1,2\} \in \mathcal{C}^*(T_1)$ and $\{1,8\} \in \mathcal{C}^*(T_2)$. Matching those two clades is better than keeping them unassigned, since:

$$\delta_{sym}(\{1,2\}, \{1,8\}) = 2 < 4 = \delta_{sym}(\{1,2\}, \emptyset) + \delta_{sym}(\emptyset, \{1,8\})$$

Resulting in a matching of overall costs of 8.

Another way to compute the distance is to use Lemma 5.6. In Figure **??** we can investigate the graph G_{T_1,T_2} with the corresponding edge weights. The optimal matching in G_{T_1,T_2} can be seen very easily. For every edge (C_1, C_2) in the matching M^*, the weight $\omega(C_1, C_2)$ is maximal among all edges starting at C_1. Thus M^* indeed is optimal.

Now let's compare this distance to the distance of T_1 and T_2 to T_3 respectively:

$$\begin{aligned} \bar{d}_{\delta_{sym}}(T_1, T_2) &= 8 \\ \bar{d}_{\delta_{sym}}(T_1, T_3) &= 19 \\ \bar{d}_{\delta_{sym}}(T_2, T_3) &= 16 \end{aligned}$$

This distance measure delivers values which are nearer to the observers intuitive answers. The distances indicate significant similarities between the trees T_1 and T_2. Furthermore they point out that T_3 has a completely different structure than the other trees.
Of course the values can only be compared with distances with respect to the same cost function δ. Generally, comparing the distances $\overline{d}_{\delta}(T_1, T_2)$ and $\bar{d}_{\delta'}(T_1, T_2)$ is not unfeasible without adding any context. For example com-

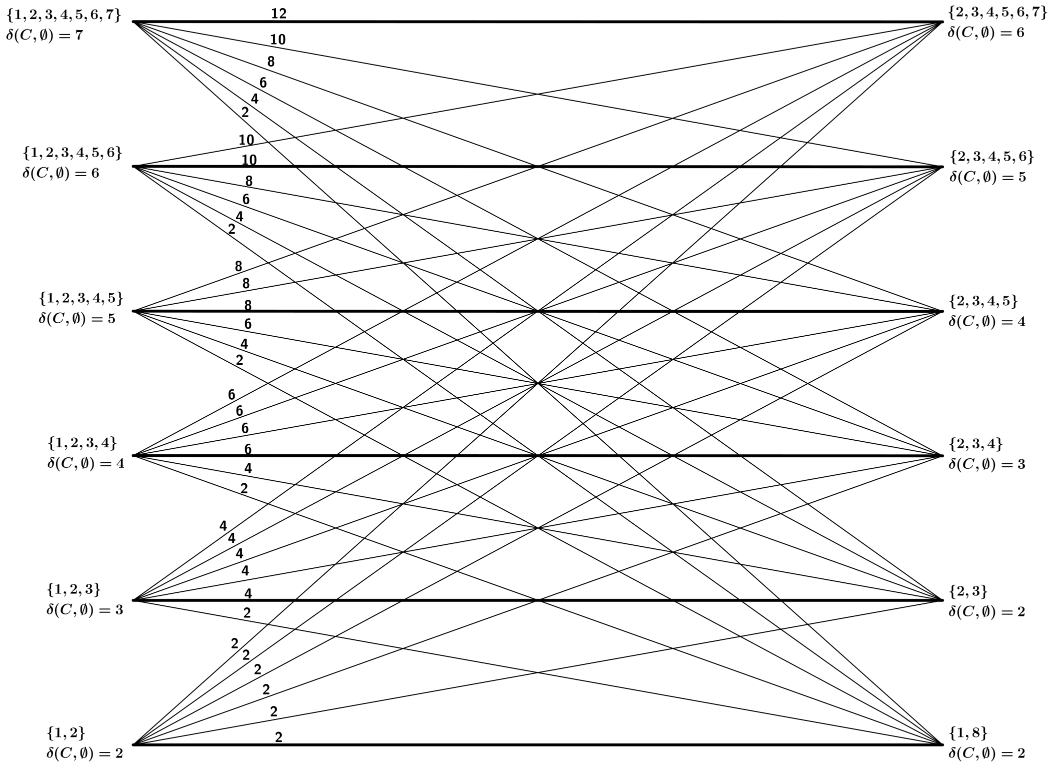

$\{1,2,3,4,5,6,7\}$
$\delta(C,\emptyset) = 7$
$\{1,2,3,4,5,6\}$
$\delta(C,\emptyset) = 6$
$\{1,2,3,4,5\}$
$\delta(C,\emptyset) = 5$
$\{1,2,3,4\}$
$\delta(C,\emptyset) = 4$
$\{1,2,3\}$
$\delta(C,\emptyset) = 3$
$\{1,2\}$
$\delta(C,\emptyset) = 2$
12
10
8
6
4
2
10
10
8
6
4
2
8
8
8
6
4
2
6
6
6
6
4
2
4
4
4
4
4
2
2
2
2
2
2
2
$\{2,3,4,5,6,7\}$
$\delta(C,\emptyset) = 6$
$\{2,3,4,5,6\}$
$\delta(C,\emptyset) = 5$
$\{2,3,4,5\}$
$\delta(C,\emptyset) = 4$
$\{2,3,4\}$
$\delta(C,\emptyset) = 3$
$\{2,3\}$
$\delta(C,\emptyset) = 2$
$\{1,8\}$
$\delta(C,\emptyset) = 2$

Figure 5.3: This is the graph G_{T_1,T_2}. On the left hand side we see nodes corresponding to the non-trivial clades of T_1, on the right hand side nodes corresponding to the ones of T_2. The edges of G_{T_1,T_2} have their weights assigned. The maximum cost matching M^* is represented by the bold edges.

paring $\bar{d}_{\delta_{RF}}(T_1, T_2) = 6 < 8 = \bar{d}_{\delta_{sym}}(T_1, T_2)$ doesn't yield any information. However one can get information from the follwoing inequalities:

$$6 = \delta_{RF}(T_1, T_2) = \delta_{RF}(T_1, T_3) = \delta_{RF}(T_2, T_3) = 6$$
$$8 = \bar{d}_{\delta_{sym}}(T_1, T_2) < 16 = \bar{d}_{\delta_{sym}}(T_2, T_3) < \bar{d}_{\delta_{sym}}(T_1, T_3) = 18$$

The minimum matching for $\bar{d}_{\delta_{sym}}(T_1, T_2)$ shown above demonstrates a problem with this straight forward approach: The clade $\{1, 2\} \in \mathcal{C}^*(T_1) \subset \{1, ..., j\} \in \mathcal{C}^*(T_1) \forall j \geq 3$, however this doesn't hold for its matched clade $\{1, 8\}$. Therefore we need to concentrate on arboreal matchings:

Definition 5.7. Let two rooted phylogenetic trees T_1, T_2 on the same set of taxa X be given. A matching M on their sets of non-trivial clades is called *arboreal* if for any two edges $\{C_1, C_2\}, \{C_1', C_2'\} \in M$ one of the following cases hold:

1. $C_1 \subseteq C_1' \wedge C_2 \subseteq C_2'$
2. $C_1' \subseteq C_1 \wedge C_2' \subseteq C_2$
3. $C_1 \cap C_1' = \emptyset \wedge C_2 \cap C_2' = \emptyset$

Definition 5.8. Let T_1 and T_2, two rooted phylogenetic trees on the same set of taxa X, and a cost function δ be given. We define $d_\delta(T_1, T_2)$ as follows:

$$d_\delta(T_1, T_2) = \min_{\substack{M \text{ matching in } G_{T_1, T_2} \\ M \text{ arboreal}}} d_\delta(M)$$

We denote the value $d_\delta(T_1, T_2)$ as the *generalized Robinson-Foulds distance* between T_1 and T_2 with respect to δ.

Remark. Some notes about the generalized Robinson-Foulds distance:

1. From now on we will abbreviate the generalized Robinson-Foulds distance with **gRF**.
2. The gRF $d_\delta(T_1, T_2)$ and the previous distance measure $\overline{d}_\delta(T_1, T_2)$ are very similar. The only difference is, that the gRF minimizes over the arboreal matchings only.
3. The inequality $\overline{d}_\delta(T_1, T_2) \leq d_\delta(T_1, T_2)$ trivially holds.

4. Lemma 5.6 also holds for the gRF $d_\delta(T_1, T_2)$. The proof works the same way, but instead of maximizing over all matchings, we only consider arboreal ones.

Lemma 5.9. *Let X, a set of taxa, and a cost function δ be given. Furthermore assume δ to be a metric. Then the gRF with respect to δ is a metric on the set of rooted phylogenetic trees on X.*

The proof is provided in the full version of the Böcker et al.'s paper [3].

5.3.1 Jaccard-Robinson-Foulds metric

One specific metric used as a cost function is motivated by the Jaccard index of two sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Generalizing this idea leads to the Jaccard weights of order k:

$$\delta_k(C_1, C_2) = \begin{cases} 0 & \text{if } C_1 = C_2 = \emptyset \\ 1 - (\frac{|C_1 \cap C_2|}{|C_1 \cup C_2|})^k & \text{else} \end{cases} \tag{5.2}$$

Lemma 5.10. *Let $\delta_k(C_1, C_2)$ be given as defined in Equation (5.2). Then $\delta_k(C_1, C_2)$ converge to the inverse Kronecker delta as $k \to \infty$:*

$$\delta_k(C_1, C_2) = \begin{cases} 0 & \text{if } C_1 = C_2 \\ 1 & \text{if } C_1 \neq C_2 \end{cases}$$

Proof. **Case 1**: $C_1 = C_2$.

$$\delta_k(C_1, C_2) = 1 - (\frac{\overbrace{|C_1 \cap C_2|}^{|C_1|}}{\underbrace{|C_1 \cup C_2|}_{|C_1|}})^k = 1 - (1)^k = 0\ \forall k \in \mathbb{N}$$

Case 2: $C_1 \neq C_2$

$$\begin{aligned}
&\Rightarrow (C_1 \cup C_2) \setminus (C_1 \cap C_2) \neq \emptyset \\
&\Rightarrow |C_1 \cup C_2| > |C_1 \cap C_2| \\
&\Rightarrow \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} < 1 \\
&\Rightarrow (\frac{|C_1 \cap C_2|}{|C_1 \cup C_2|})^k \underset{k \to \infty}{\to} 0
\end{aligned}$$

□

Definition 5.11. Let T_1 and T_2, two rooted phylogenetic trees on the same set of taxa X and a positive $k \in \mathbb{R}_{\geq 0}$ be given. The induced gRF metric is called Jaccard-Robinson-Foulds metric (**JRF**) of order k and is denoted by $d_{JRF}^{(k)}(T_1, T_2)$.

Theorem 5.12. *The JRF of order k converges to the Robinson-Foulds distance as* $k \to \infty$.

Proof. For a fixed problem, there is only a finite number of non-trivial clades. Every Jaccard-distance tends to the inverse Kronecker delta as stated in Lemma 5.10. Therefore the following holds:

$$\forall\, 0 < q < 1 : \exists\, K \in \mathbb{N} \text{ s.t.} \forall\, k > K :$$

$$\delta_k(C_1, C_2) \begin{cases} = 0 & \text{if } C_1 = C_2 \\ = 1 & \text{else if } C_1 = \emptyset \text{ or } C_2 = \emptyset \\ > 1 - q & \text{else if } C_1 \neq C_2 \end{cases}$$

Let M be a matching that minimizes $d_{RF}(T_1, T_2)$.

$$d_{RF}(M) - d^{(k)}_{JRF}(M) =$$

$$\sum_{(C,C')\in M} (\delta_{RF}(C,C') - \delta^{(k)}_{JRF}(C,C')) + \sum_{\substack{C\in\mathcal{C}^*(T_1),\\ C \text{ unmatched in } M}} (\overbrace{\delta_{RF}(C,\emptyset)}^{=1} - \overbrace{\delta^{(k)}_{JRF}(C,\emptyset)}^{=1})$$

$$+ \sum_{\substack{C'\in\mathcal{C}^*(T_2),\\ C' \text{ unmatched in } M}} (\overbrace{\delta_{RF}(\emptyset,C')}^{=1} - \overbrace{\delta^{(k)}_{JRF}(\emptyset,C')}^{=1}) =$$

$$\sum_{\substack{(C,C')\in M\\ C=C'}} (\overbrace{\delta_{RF}(C,C')}^{=0} - \overbrace{\delta^{(k)}_{JRF}(C,C')}^{=0}) + \sum_{\substack{(C,C')\in M\\ C\neq C'}} (\overbrace{\delta_{RF}(C,C')}^{=1} - \overbrace{\delta^{(k)}_{JRF}(C,C')}^{>1-q}) <$$

$$\sum_{\substack{(C,C')\in M\\ C\neq C'}} q < n * q \underset{q\to 0}{\to} 0$$

□

5.3.2 Computational Complexity

In their paper Böcker et al. demonstrate a polynomial reduction from the $(3, 4)$-SAT problem to the problem of finding a minimal cost arboreal matching, even if the cost function δ is a metric. The $(3, 4)$-SAT is the problem of finding a satisfying assignment for a Boolean formula in which every clause consists of exactly 3 literals and any variable occurs 4 times. The authors of the above mentioned paper were able to construct a minimum arboreal matching instance I for any Boolean formula $\phi \in (3, 4)$-SAT. If the problem I, using the symmetric difference as cost function, admits a solution with a value smaller than a certain value, the original problem of finding a correct assignment for ϕ has a solution. For more details we suggest to read the original paper [3].

Theorem 5.13. *For an instance of the arboreal matching using the symmetric difference as cost function δ and an integer k, it is NP-complete whether there exists an arboreal matching of cost at most k.*

5.3.3 An Integer Linear Program

One way to approach an $\mathcal{NP}$-complete problem is to formulate it as an integer linear program and solve the latter. Therefore Böcker et al. set up an integer linear programming formulation to find a minimum cost arboreal matching.

Let two rooted phylogenetic trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ and a cost function $\delta : \mathcal{C}(T_1 \times \mathcal{C}(T_2) \mapsto \mathbb{R}_{\geq 0}$ be given. We number the clades in $\mathcal{C}(T_i)$ from 1 to $|V_i|$ for $i \in \{1, 2\}$. Then the indicator variable $x_{i,j}$ represents whether the i-th clade C_i in $\mathcal{C}(T_1)$ is matched with the j-th clade C'_j in $\mathcal{C}(T_2)$. We use the cost function introduced in Equation (5.1) to find a minimum cost matching while maximizing the sum of the cost function. To recapitulate, the value for $\omega(C_i, C'_j)$ is:

$$\omega(C_i, C'_j) = \delta(C_i, \emptyset) + \delta(\emptyset, C'_j) - \delta(C_i, C'_j)$$

We also have to make sure that the assignment we receive satisfies the conditions for an arboreal matching. Therefore we introduce the set $\mathcal{I}$ with the following properties:

$$\mathcal{I} := \left\{ \{(i,j),(k,l)\} \;\middle|\; \begin{array}{l}\text{The edges } (C_i, C'_j) \text{ and } (C_k, C'_l) \text{ violate the conditions} \\ \text{for arboreal matchings defined in Definition 5.7}\end{array} \right\}$$

Theorem 5.14. *Let T_1 and T_2, two rooted phylogenetic trees on the same set of taxa X, and a cost function δ be given. Furthermore let a fixed order of all non-trivial clades in $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$ and the cost function $\omega(C_i, C'_j)$ be given. We introduce* *a linear integer problem with decision variables $x_{i,j}$ as follows.*

$$\max \sum_{i=1}^{|V_1|} \sum_{j=1}^{|V_2|} \omega(C_i, C'_j) x_{i,j} \tag{5.3}$$

$$s.t. \sum_{j=1}^{|V_2|} x_{i,j} \leq 1 \qquad \forall i \in \{1, ..., |V_1|\} \tag{5.4}$$

$$\sum_{i=1}^{|V_1|} x_{i,j} \leq 1 \qquad \forall j \in \{1, ..., |V_2|\} \tag{5.5}$$

$$x_{i,j} + x_{k,l} \leq 1 \qquad \forall \{(i,j),(k,l)\} \in \mathcal{I} \tag{5.6}$$

$$x_{i,j} \in \{0,1\} \qquad \forall i \in \{1, ..., |V_1|\}, \forall j \in \{1, ..., |V_2|\} \tag{5.7}$$

Let $x^*_{i,j}$ be the values of the indicator variables $x_{i,j}$ of an optimal solution of the linear integer problem. Let M^* be the following set of edges:

$$(C_i, C'_j) \in M^* \quad \Leftrightarrow \quad x^*_{i,j} = 1$$

Then the set of edges M^* is a matching that realizes the value of the gRF:

$$d_\delta(T_1, T_2) = d_\delta(M^*)$$

Proof. Here we sketch the proof of the equivalence between the linear integer program (5.3) - (5.7) and the minimum cost arboreal matching problem. First and foremost, the restriction to integer decision variables in Equation (5.7) ensures, that we either choose an edge to be in the matching or not. Thus we indeed get a set of edges. Furthermore Equations (5.4) and (5.5) restricts the chosen set of edges to be a (not necessarily complete) matching. Last but not least, because of Equation (5.6) we can be certain that the matching we end up with is a an arboreal matching, since we constructed the set $\mathcal{I}$ exactly this way. As proven in Lemma 5.6, the arboreal matching of maximum cost with respect to the cost function (5.3) is an arboreal matching that minimizes the gRF $d_\delta(T_1, T_2)$. □

6 Implementation Generalized Robinson Foulds

In this chapter we discuss the implementation for the Generalized Robinson Foulds distance. First we discuss the necessary preparation and some additional comments. These include the creation of test instances and choosing meaningful distance function. We proceed with details about the implementation and end this chapter with a short summary of the results.
All programs, functionalities are programmed in the programming language Python. We created a module to compute and store the Catalan numbers, create and store randomized full binary trees and finally pairwise comparing them. All implementation details can be found in my Github repository [8].

6.1 Preparation and Overview

The comparison of tree distances is, as already explained, a necessary tool for different research fields. But since we have to be able to find test instances that fit into the environment of the tree edit distance as well as into the one of the Robinson-Foulds distance, we are restricted to certain kinds of test data. The Robinson-Foulds distance only makes sense in the setting of phylogenetic trees on the same set of taxa, as the inner nodes aren't labelled and don't get any attention. Therefore this needs to be reflected in the tree edit distance as well. Furthermore we only take full binary trees into consideration. These are trees where every inner node has exactly two children. On the one hand this excludes multifurcations, on the other hand it ensures that any inner node represents a split between subsets of taxons.

6.1.1 Creating Test Instances

Since we hadn't found a suitable database of test instances, we needed to create it on our own. As already discussed, we have to have a set of full binary trees with pairwise differently labelled leaves. The test set needs to be a randomly selected subset of full binary trees with n leaves, $n \in \mathbb{N}$. Therefore we created an algorithm that chooses a full binary tree uniformly distributed which is based on the commonly known fact that the number of full binary trees with n leaves is the $(n-1)$-th catalan number. Recursively, we choose the sizes of the left and the right subtrees for every inner node. In each step we need to make sure that we choose the sizes such that every possible outcome is equally possible. Before providing the algorithm, we explain the main idea with the first few recursion steps:

- $\underline{n=2}$: It is obvious that there is only one full binary tree with two leaves. This also fits the Catalan number $C_1 = 1$.
- $\underline{n=3}$: This case is trivial as well. For the root we can decide whether the left subtree has one or two leaves. This automatically determines both subtrees. Therefore there are exactly two possible full binary trees with three leaves.
- $\underline{n=4}$: We start with the root again. Its left subtree may contain between one and three leaves. Let's further discuss this case distinction to give an idea about the algorithm later on:
  - Case 1: The left subtree contains one leaf.
    Then the left subtree needs to be a full binary tree with one leaf. The number of such trees corresponds to $C_0 = 1$. Furthermore the right subtree needs to contain three leaves. There are $C_2 = 2$ possibilities of such trees. So we end up with two different full binary trees where the left subtree of the root has only one leaf.

- Case 2: The left subtree contains two leaves
  Thus the same has to hold for the right subtree. Both of them are determined since there is only $C_1 = 1$ such tree.
- Case 3: The left subtree contains three leaves.
  This case is symmetric to the Case 1, so there are two such trees.

This results in a total number of $C_3 = 5$ full binary trees with four leaves.

Our algorithm needs to make sure that choosing the number of leaves of the left and right subtrees of the root happens with correct probabilities to ensure uniform distribution among all possible full binary trees.

We exemplify the proof of uniform distribution for $n = 4$ once again. Every subtree has to be chosen with probability of $\frac{1}{5} = \frac{1}{C_3}$. Trivially we have to select the Case 2, where both subtrees contain two leaves with probability $\frac{1}{5}$ since this already determines one possible outcome. The other cases are symmetric, so both of them need to be chosen with equal possibility of $\frac{2}{5}$. In both cases we have to further choose a full binary tree with three leaves. Since there are exactly two of them, we just have to make sure that both of them are chosen with probability $\frac{1}{2}$. Thus we are able to choose each full binary tree with four leaves with equal possibility.

Lemma 6.1. *Algorithm 2 returns a full binary tree with n leaves, where any such tree is chosen with the same probability.*

Proof. We make an inductive argument:
Induction Base: The routine CreateFullBinaryTree(1) returns the only possible full binary tree with one leaf. Therefore it also gets chosen with probability 1.
Induction Hypothesis: CreateFullBinaryTree(j) returns a full binary tree with j leaves chosen uniformly at random among all such trees for all $j < n$.
Induction Step: $n - 1 \to n$. We can assume that $n > 1$, so the algorithm prepares the recursive step. First it computes some intervals. We define $p_i := \frac{C_{i-1}C_{n-1-i}}{C_{n-1}}$ as the fraction inside the for loop.

Algorithm 2 Choosing a full binary tree with n leaves with equal probability

```
function CreateFullBinaryTree($n$)
    if $n == 1$ then
        return A single node
    else
        var $p$ = 0;                                        ▷ Probability for every case
        var $P$ = 0;                                        ▷ Sum of probabilities until now
        list $I$ = [];                                      ▷ List of intervals between 0 and 1
        for $i = 1, (n-1)$ do
            $p = \frac{C_{i-1}C_{n-1-i}}{C_{n-1}};$
            $I[i] = [P, P+p];$
            $P = P + p;$
        end for
        $r \in [0,1)$ chosen uniformly at random;
        Let $j$ be the index for which $r$ lies in $I[j]$;
        Let $L$ = CreateFullBinaryTree($j$);
        Let $R$ = CreateFullBinaryTree($n-j$);
        return The binary tree with the root having the roots of $L$ and $R$ as its left and right children respectively;
    end if
end function
```

Claim 1: $P = \sum_{i=1]}^{n-1} p_i = 1$
Proof of Claim 1:

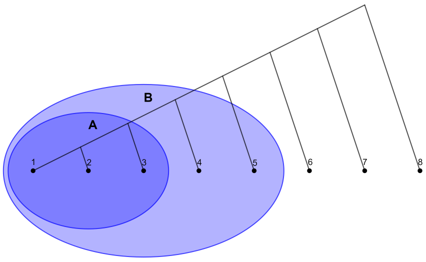
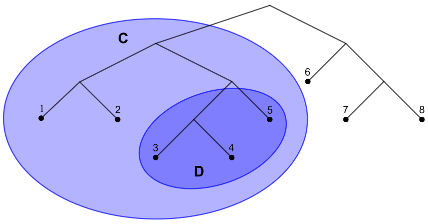
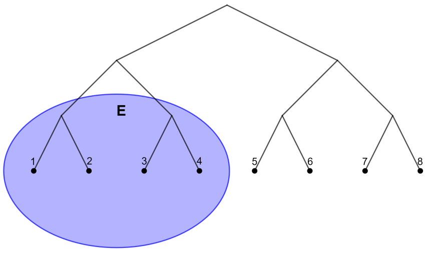
$$P = \sum_{i=1}^{n-1} p_i = \sum_{i=1}^{n-1} \frac{C_{i-1}C_{n-1-i}}{C_{n-1}} = \frac{\sum_{i=1}^{n-1} C_{i-1}C_{n-1-i}}{C_{n-1}} = \frac{\overbrace{\sum_{j=0}^{(n-1)-1} C_j C_{(n-1)-1-j}}^{C_{n-1}}}{C_{n-1}} = 1$$

Thus we can talk about the p_i's as probabilities. The next step in the algorithm is to choose r uniformly at random, which lies within the interval $I[i]$ with probability p_i forall $1 \leq i < (n-1)$. The routine recursively creates the left subtree of the root with i leaves and a right subtree with $n-i$ leaves. Therefore the probability is p_i that we create a full binary tree, where the left subtree of the root has i leaves and the right one has $n-i$ leaves. By the induction hypothesis, we know that CreateFullBinaryTree(i) chooses a full binary tree with i leaves uniformly at random. There are C_{i-1} such trees, so any one of them gets chosen with probability $\frac{1}{C_{i-1}}$, analogously for $n-i$.
Take an arbitrarily chosen full binary tree with n leaves. There exists an $1 \leq j < n$ s.t. the tree has j leaves in the left subtree of the root and $n-j$ leaves in the right subtree. The probability is $p_j = \frac{C_{j-1}C_{n-1-j}}{C_{n-1}}$ that the algorithm creates a tree with these properties. Furthermore, using the induction hypothesis, it returns the left subtree with probability $\frac{1}{C_{j-1}}$ and the right one with probability $\frac{1}{C_{n-1-j}}$. All in all, the algorithm returns this arbitrary full binary tree with the following probability:

$$\underbrace{\frac{C_{j-1}C_{n-1-j}}{C_{n-1}}}_{\substack{\text{correct number of}\\ \text{leaves in}\\ \text{both subtrees}}} \cdot \underbrace{\frac{1}{C_{j-1}}}_{\substack{\text{returning correct}\\ \text{left subtree}}} \cdot \underbrace{\frac{1}{C_{n-1-j}}}_{\substack{\text{returning correct}\\ \text{right subtree}}} = \frac{1}{C_{n-1}}$$

□

Performing this routine repeatedly allowed us to construct a uniformly randomized set of full binary trees with different numbers of leaves. We stored the test instances as Json-encoded lists of trees. We represented a tree as a recursive list with one or two elements. Take a look at the following examples:

Figure of tree	Python List
	$\Bigg[\Big[\underbrace{\big[\overbrace{[[[1],[2]],[3]]}^{A},[4]],[5]\big]}_{8},[6]],[7]\Big],[8]\Bigg]$
	$\Bigg[\underbrace{\Big[\big[[1],[2]\big],\overbrace{\big[[[3],[4]],[5]\big]}^{D}\Big]}_{C},\Big[[6],\big[[7],[8]\big]\Big]\Bigg]$
	$\Bigg[\underbrace{\Big[\big[[1],[2]\big],\big[[3],[4]\big]\Big]}_{E},\Big[\big[[5],[6]\big],\big[[7],[8]\big]\Big]\Bigg]$

6.1.2 Distance Function

The generalized Robinson-Foulds allows us to choose different distance measures. But since we have to compare our results with the tree edit distance, we need to choose a distance that counts wrong leaves. Therefore we stick with the introduced Robinson-Jaccard metric. We will compute the distances between trees with different values for the constant k. Thus we may see a pattern.

6.2 Implementation Details

We based our implementations of both the generalized Robinson-Foulds distance and the tree edit distance on the data structure used in an implementation of Shasha and Zhangs algorithm [10] introduced in Section 3.3.1. For computing the generalized Robinson-Foulds distance we extended the data structure *Node* of the above mentioned git repository to include the following functions:

```python
from zss.simple_tree import Node

class ExtendedNode(Node):
    def get_list(self):
        #recursive function that returns a nested list representing the tree structure
        #used to create example trees

    def number_of_leaves(self):
        #returns the overall number of leaves in the tree

    def list_of_leaves(self):
        #returns a list of the leaf-nodes (objects) in the tree

    def list_of_leaf_labels(self):
        #returns a list of the labels (string) of the leaves in the tree

    def get_clusters(self, exclude_leaf_labels = 0):
        #returns a list of the clusters in this tree

    def label_leaves_randomly(self):
        #during construction of tree examples we first construct trees and afterwards label them randomly
```

Listing 6.1: Scratch of the class ExtendedNode

We based the computation of the generalized Robinson Foulds distance on the linear program introduced in Theorem 5.14. We used a widely known, open-source library for linear programms within python The **PuLP**package.

Therefore we just had to compute the distance matrix of C and C' and formulate the restrictions accordingly.

6.3 Results

As mentioned above we used the Jaccard metric to calculate the distance between sets of clusters. We used $k \in \{1, 2, 4, 8, 16, 64\}$ and saw a pattern: Not only is the distance directly proportional to the value of k, but the overall distance tends to the standard Robinson Foulds distance very quickly as k increases.

Furthermore, without regarding the quality of the results, we realized that the running time is a big problem for this approach. Handling instances with 24 leaves on both trees already takes 500 seconds. Increasing the number of leaves to 32 on each tree leads to a running time of about 3100 seconds. It already takes 22 to create the complete distance matrix, however one has to be very patient to wait for a solution.

It is possible that more evolved linear programming packages solve these problems more efficiently. Nevertheless, this approach doesn't seem to bring fast solutions for big problems.

7 Implementation Tree Edit Distance

In this chapter we present the implementation of Shasha and Zhangs algorithm by Henderson [10]. We discuss meaningful choices of cost functions and compare them with each other. Last but not least we compare the tree edit distance's advantages and disadvantages with respect to the Generalized Robinson Foulds distance.

7.1 Shasha and Zhang's algorithm by Henderson

Suppose we want to compute the tree edit distance for two trees A and B. Henderson's implementation can be split into two separated important steps:

1. Finding out the post-order traversal index and the keyroots for the two trees under consideration.
2. Computing the tree edit distances for all relevant subproblems, i.e. combinations of subtrees induced by keyroots of A and B respectively.

Ad Step 1:

In Listing 7.1 we present the initialization of the class AnnotatedTree. It can be split up into two parts: Step 1a) and Step 1b). We will give a short summary of what happens in these two substeps later on.

```
class AnnotatedTree(object):
    def __init__(self, root, get_children):
        #initialization of class properties:
        #nodes, keyroots, ids, stack, pstack

        #********** Step 1a) *********
        stack.append((root, collections.deque()))
        j = 0
        while len(stack) > 0:
            n, anc = stack.pop()
            nid = j
            for c in self.get_children(n):
                a = collections.deque(anc)
                a.appendleft(nid)
                stack.append((c, a))
            pstack.append(((n, nid), anc))
            j += 1
        #********** Step 1b) *********
        lmds = dict()
        keyroots = dict()
        i = 0
        while len(pstack) > 0:
            (n, nid), anc = pstack.pop()
            self.nodes.append(n)
            self.ids.append(nid)
            if not self.get_children(n):
                lmd = i
                for a in anc:
                    if a not in lmds: lmds[a] = i
                    else: break
            else:
                try: lmd = lmds[nid]
                except:
                    import pdb
                    pdb.set_trace()
            self.lmds.append(lmd)
            keyroots[lmd] = i
            i += 1
        self.keyroots = sorted(keyroots.values())
```

Listing 7.1: Initialization of an AnnotatedTree

Ad Step 1a):
This substep initializes a stack *pstack* that is needed for the Step 1b). But first we take a close look at the variable called *stack*. A stack is a specific data structure. It is a collection of objects that supports fast last-in, first-out semantics. Throughout the process of the loop, the stack always consists of a pair of data, namely a node and a list of all its ancestors. Therefore we initialize the stack with the pair of the annotated tree's root and an empty collection, since the root doesn't have an ancestor.
While the stack still contains some pair of data we perform a function on it called *pop*. This function returns the last element of the stack and removes it from the stack. We associate every node with a unique id *nid*. If the inspected node has some children we enlarge the stack with pairs of each children and the updated list of ancestors. The essential detail is that we *append* this pair to the stack, which means that we put it on the end of the stack. The reason why this is so important will be explained later. After appending all children to the stack, the node will be appended to another stack called *pstack* which will be used in Step 1b), together with its node id *nid* and the list of ancestors. Let's go through the process for the first pair of data, namely the root and the empty set of ancestors. We assign the *nid* 0 to the root and go through its children. Here we first append the root's left child, afterwards its right child. Thus the right child is further in the back of the stack than the left child. Since we only *pop* the stack, the right child will be out of the stack earlier than the left child. Furthermore, during the next loop we append the stack with other nodes again, pushing the left child of the root further down the stack. Thus we end up with the new stack *pstack* that has the correct left to right post traversal ordering. The further back a node is in the stack *pstack*, the smaller is its post traversal ordering index.
Ad Step 1b):
In this step we determine the keyroots of the investigated tree. The key idea is, that a node is a keyroot if and only if it is the node with the highest post-order traversal index among all nodes with the same left most descendant. The right sibling of a node *n* has a different left most descendant than the node *n* itself. But any node with a higher index can not have the same left

most descandant as this right sibling because of the way we indexed the nodes. Thus if we determine the node with the highest index among all nodes that share the same leftmost descendant, then we know that it has to be a keyroot. For this we need two temporary dictionaries *lmds*, the list of the left most descendant for every node, and *keyroots*, a dictionary that saves the highest index of all nodes that share a leftmost descendant.
As previously discussed we go through *pstack*, a stack of nodes in the correct order. The counter *i* saves the actual post order index, it increments after every loop. During every loop we pop the stack and get a node *n*, its node id *nid* and its ancestors. Then we check whether the node is a leaf or not. If it indeed is, then we set the left most descendant of every ancestor to this node, as long as it does not already have a left most descendant. If it is an interior node, then we just get the left most descendant by checking the dictionary *lmds*. The most important step happens near the end of the loop: setting $keyroots[lmd] = i$. In this way we find the highest index *i* of all nodes, that share the same leftmost descendant. Therefore, after having finished the routine, we get the keyroots by checking the values of the temporary dictionary *keyroots*.

This was a detailed description of the way an object of type AnnotatedTree gets initialized. It makes use of efficiently appending and popping a stack to receive the correct indexing and the all the necessary information to get the list of keyroots.

Ad Step 2:
After the initialization of the AnnotatedTree, the actual computation of the tree edit distance is done by first considering all relevant subproblems. A distance matrix is created that stores the pairwise distance between all relevant subproblems of the two investigated trees. This matrix gets built incrementally by applying the recursion stated in Lemma 3.6. The technical details can be seen in the actual implementation [10].

7.2 Distance measures

The most simple distance measure for the tree edit distance just counts the number of operations needed to transform one tree into the other. This implies that any insertion, any deletion and any renaming costs the same value of 1. We call this distance measure the *standard tree edit distance* (STED) The big difference between the tree edit distance and the generalized Robinson Foulds is that the latter does not take the interior nodes of the trees into account. Yet every insertion and deletion of an interior node increases the tree edit distance. Therefore the first adaption on the operation costs we did was to make the insertion and deletion of interior nodes free of charge. We will later see how this influences the overall comparison between these two distances. This distance measure will later be referred to as *cheap tree edit distance* (CTED)
Another distinction is that the Robinson Foulds distance doesn't make a difference between left and right. Therefore there was an attempt to improve the distance measure by adapting the direction. The idea is to change one of the trees by swapping the order among some sibling pairings. Considering all combinations of sibling pairings leeds to $O(2^n)$ possibilities of swapping siblings, which is obviously not a good solution. To explain our approach we consider the two roots r_1 and r_2 of the inspected trees. Let's denote the left subtree of the root r_i as L_i and the right one as R_i for $i \in \{1,2\}$. If $|L_1| > |R_1|$, meaning that L_1 has more leaves than R_1, but the opposite holds true for the other tree, i.e. $|L_2| < |R_2|$, then we swap the children of r_1. The goal is to make the two trees as equally distributed as possible without changing the set of clusters for these randomly created trees. Thus the Robinson Foulds distance stays the same, but the tree edit distance may change. After having investigated the root we perform the same comparison between l_1 and l_2, r_1 and r_2 respectively. If we adapt one of the trees to make them as equally distributed as possible before calculating, we add the adjective *adapted* to the distance. So later on we will refer to it as the *adapted standard tree edit distance* (aSTED) and the *adapted cheap tree edit distance* (aCTED).

Furthermore we wanted to investigate a compromise bewteen the CTED and the STED by varying the costs of insertion and deletion of inner nodes. We introduced a variable $0 < \delta < 1$ for the costs of insertion and deletion. Assuming $\delta = \frac{1}{2}$, we call this distance measure $\frac{1}{2}$-*delta tree edit distance* ($\frac{1}{2}$-DTED).

7.3 Results

For smaller trees we were able to compare the results with the ones from the generalized Robinson Foulds distance. But for trees with more than 32 leaves it was not possible to solve the linear program, as this would have taken too much time. However we were able to compute the tree edit distance for trees with up to 512 leaves.

Obviously, we came to the conclusion that the STED lead to bigger distance values than the CTED. This is trivial as every operation is as least as expensive for the STED in comparison to the CTED. Furthermore, to get a better feeling, we put the distances in relation to the number of leaves. From now on we denote the number of leaves on each tree as the variable n. We discovered two trends as n is increasing:

- The average of the relation of the CTED to n is diverging to 1.
- The average of the relation of the STED to n among all examples is strongly increasing.

The explanation for the first trend is rather intuitive. Figure 7.1 indicates that any tree has distance $\leq n$ to any other tree on the same number of leaves n. Since the CTED converges to n it seems as if this distance is not a good measure of similarities between structure of the two trees at all, but only whether the order of the leaves' labels is similar.
Thus we investigated the DTED, mainly with $\delta = \frac{1}{2}$, and observed some interesting behaviors. For example, The distribution of the relation of the $\frac{1}{2}$-DTED to n becomes more and more constant as n is increasing.

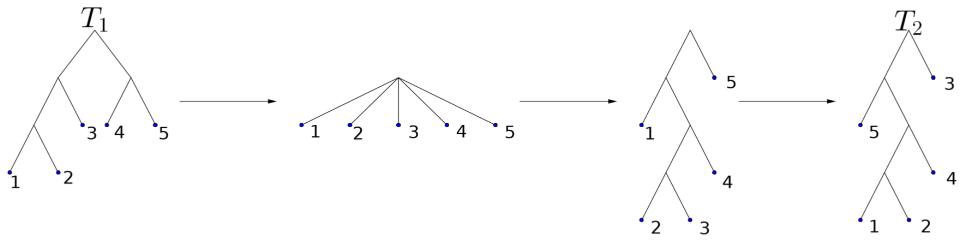


Figure 7.1: A step guide for limiting the CTED. First we delete all inner nodes until all leaves are directly connected to the root. Afterwards we insert all inner nodes until the structure is the same as the one of tree T_2. Now we only have to rename all leaves accordingly. Since the deletion and insertion of inner nodes is for free we end up with a CTED $\leq n$.

7.4 Comparing Results

8 Conclusion

This thesis tries to give a short insight into a large and widely ranged topic of comparing trees.

First we give some basic notations and definitions since the language varies greatly between different papers and authors.

The first idea of comparing trees is the tree edit distance. Editing one tree into the other, by using simple operations such as deleting, inserting and relabelling step by step, is one intuitive way of comparing trees. Every operation is associated with some costs. The task is to find a sequence of editing operations of minimal costs, that alters both trees to become the same. There are simple algorithms using a dynamic programming approach that can compute the tree edit distance to optimality. We start with Shasha and Zhang [14], who used a trivial decomposition strategy which guides an iterative, recursive algorithm which always compares the rightmost subtrees. Then we present the algorithm of Klein [11] and the algorithm of Demaine et al. [6]. These algorithms involve more sophisticated decomposition strategies which take into account the sizes of the investigated subtrees. Demaine et al. provided a lower bound on the running time for computing the tree edit distance with a dynamic programming approach. Furthermore they proved that their algorithm satisfies this bound, making it optimal among all algorithms with a dynamic programming approach.

Moreover in chapter 4 we proceed with the so called flexible tree matching. Here the idea is that one may relax the restrictions of ancestry and sibling groups. Instead of forbidding such violations we just have to pay a fine for any occurrence. Unfortunately finding the flexible tree edit distance is a strongly $\mathcal{NP}$-complete problem. Nevertheless we present a model for approximation heuristics. Kumar et al. [12] provided a Monte Carlo algorithm to compute an approximation of the flexible tree matching.

In chapter 5 we take a look at a widely used measure, the Robinson-Foulds metric. Working on the same set of taxa (or leaves), we want to know how many clades are present in exactly one of the two investigated trees. We discuss its advantages and disadvantages and generalize it to make it more applicable using more evolved cost functions. However, we still end up with a $\mathcal{NP}$-complete problem of finding a minimum cost arboreal matching between the sets of non-trivial clades.

Bibliography

[1] Apostolico A., Galil Z., *Pattern matching algorithms*, Oxford University Press, 1997

[2] Bille P, *A survey on tree edit distance and related problems*, Theoretical computer science, 2005, 217—239

[3] Böcker S., Canzar S., Klau G., *The Generalized Robinson-Foulds Metric*, International Workshop on Algorithms in Bioinformatics, 2013, 156—169,

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

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

[6] 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

[7] 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

[8] Andritsch C., *Master Thesis*, Github Repository, `https://github.com/bananajoe/masters_thesis`, 2019

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

[10] Tim Henderson *Zhang-Shasha: Tree edit distance in Python*, Github Repository, `https://github.com/timtadh/zhang-shasha`, 2019

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

[12] Kumar R., Talton J., Ahmad S., Roughgarden T., Klemmer S., *Flexible Tree Matching*, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, 2011,

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

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

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

[16] Valiente G., *Algorithms on Trees and Graphs*, Springer-Verlag, 2002,

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