Clemens Andritsch, BSc

Comparing Trees

Master's Thesis

to achieve the university degree of

Diplomingenieur

Master's degree programme: Mathematics

submitted to

Graz University of Technology

Supervisor

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

Institute for Discrete Mathematics

Graz, June 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date Signature

Contents

1 Introduction **1**

2 Basics and notation **3**
2.1 Basic Graph Theoretic Concepts 3
2.2 Other necessary Tools 10

3 Tree Edit Distance **14**
3.1 Introduction 14
3.2 Short History of the Tree Edit Distance 15
3.3 Dynamic Programming Approach 17
3.3.1 Shasha and Zhang's algorithm 18
3.3.2 Klein's algorithm 21
3.3.3 Demaine et al.'s optimal algorithm 22
3.3.4 Lower bound on decomposition algorithms 23

4 Flexible Tree Matching **26**
4.1 The Model for the Flexible Tree Edit Distance 29
4.2 Approximation and Conclusion 33

5 Robinson Foulds Metric **34**
5.1 Additional Background 34
5.2 The original metric 35
5.3 The Generalized Robinson Foulds 37
5.3.1 Jaccard-Robinson-Foulds metric 43
5.3.2 Computational Complexity 45
5.3.3 An Integer Linear Program 45

6 Implementation generalized Robinson Foulds **53**
6.1 Preparation and Overview 53
6.1.1 Creating Test Instances 54
6.1.2 Distance Function 58
6.2 Implementation Details 59

6.3 Results 60

7 Implementation Tree Edit Distance 63

7.1 Shasha and Zhang's algorithm by Henderson 63

7.2 Distance measures 66

7.3 Results 67

8 Comparing the Tree Edit Distance and the generalized Robinson Foulds Distance 71

8.1 Time Complexity 71

8.2 Discussing the Results 72

9 Conclusion 77

Bibliography 79

1 Introduction

A tree is a special type of graph which contains exactly one path between any two nodes. This graph theoretic concept can be used to describe data in different areas of research. In most applications trees have to be compared at some point. In the area of bioinformatics there is a need to measure similarities between different RNA-structures. Computer scientists have to compare structured text databases or natural languages. The preliminaries and circumstances differ, resulting in a variety of possibilities to compare trees. Some techniques can only handle a very specific type of tree, others may work for arbitrary trees, but have disadvantages for special cases.

Therefore there is a need for a quantitative measure of how similar two trees are. Such a measure is called *distance measure*. A distance can be seen as a measure of similarities: The smaller the distance between two trees, the more similar they are. Conversely, two trees with a big distance should be quite different. The notion of similarity has to be defined.

This thesis presents multiple tools for comparing trees. It provides insights into applications and compares two seemingly very different distance measures, to see if they have similarities.
In chapter 2 basic notations and important concepts are introduced. Chapter 3 defines the *tree edit distance* and presents dynamic programming approaches to calculate it. In chapter 4 the tree edit distance is generalized to the *flexible tree edit distance*. Chapter 5 deals the *Robinson Foulds metric* which can only be applied to compare two trees of a special class, namely evolutionary trees. Chapters 6 and 7 present the implementation of different algorithms that compute the tree edit distance and a generalized Robinson

Foulds metric. Finally chapter 8 compares the results of these implementations and shows that the generalized Robinson Foulds metric can't be simulated by a tree edit distance.

2 Basics and notation

In this chapter basic definitions and notations that are used throughout the thesis are introduced. They include special property of trees, basic concepts for nodes in trees and more evolved notations that are needed in later sections of this thesis

2.1 Basic Graph Theoretic Concepts

Definition 2.1. Let $T = (V, E)$ be a (simple) undirected 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.

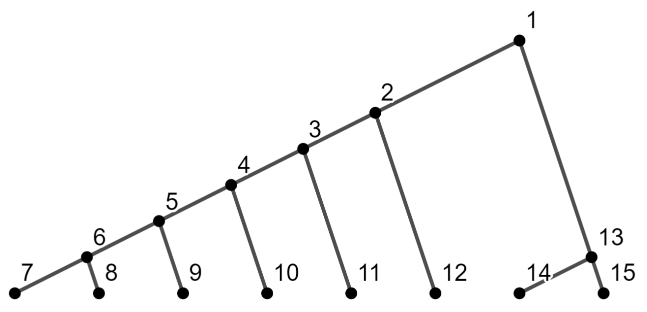


Figure 2.1: The illustration suggests, that the node 1 is the root of the tree. Together with some function $l : \ \{1, ..15\} \mapsto \{A, B, C\}$ T is a rooted labelled tree.

Definition 2.2. Let $T = (V, E)$ be a rooted labelled tree with root $r \in V$. The *parent* $P(v) \in V$ of a node $v \in V \setminus \{r\}$ is defined as the 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 $r \in V$ and $w, v \in V$. The node w is a *child* of v if and only if v is the parent of w. Furthermore the set $C_T(v)$ is defined as 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 $r \in V$ and $w \neq v \in V \setminus \{r\}$. The node w is a sibling of v if and only if $P(v) = P(w)$. The set $S(v)$ denotes the *sibling group of v*:

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

The sibling group of the root r is manually defined as $S(r) := \{r\}$.

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

$$v \in S(v)$$

This implies: $|S(v)| >= 1$.

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

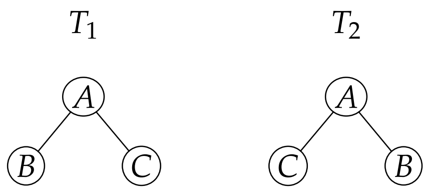


Figure 2.2: Assuming T_1 and T_2 to be unordered trees yields $T_1 = T_2$. Otherwise, considering them to be ordered like in the figure implies $T_1 \neq T_2$.

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

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
```

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

Definition 2.8. Let $T = (V, E)$ be an ordered forest. The left- and rightmost subtrees of T are denoted as L_T and R_T respectively. Furthermore the roots of L_T and R_T are denoted by l_T and r_T respectively

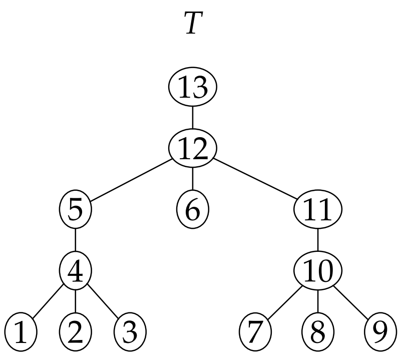


Figure 2.3: An example of the post order indexing. Note that for every subtree the root is indexed at last.

Remark. Consider the tree T in Figure 2.3. Using the notion introduced above, one can characterize the following trees:

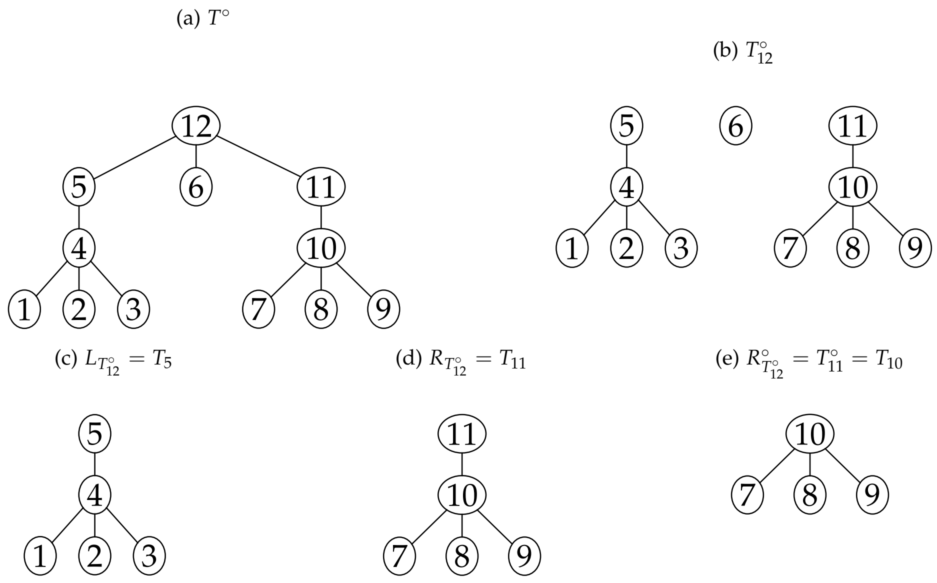

(a) T°
(b) T°_{12}
(c) $L_{T^\circ_{12}} = T_5$
(d) $R_{T^\circ_{12}} = T_{11}$
(e) $R^\circ_{T^\circ_{12}} = T^\circ_{11} = T_{10}$

Figure 2.4: Different subtrees using the introduced notion.

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

Remark. These definitions will mainly be used for stating the running times of algorithms. Furthermore, if a tree is significantly bigger, it is assumed 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 $V(T'') = \{i_{T''}, ..., j_{T''}\}$.
The forest T' is called 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. The set of *keyroots* of T is defined 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 $r_1, ...r_n$. The notion of keyroots is extended to be the union of keyroots of each individual tree T_i:

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

Definition 2.12. Let T be an ordered tree rooted at r. The *collapse depth* $\text{cdepth}(v)$ of a node v is defined as 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 one chooses a node $m \in C(n)$ among those with the most descendants in $C(n)$ arbitrarily. This node m is then defined 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. An edge is called *heavy*, if it connects a non-leaf with its heavy child. Furthermore a path connecting a light node with a leaf and only consisting of heavy edges is called a *heavy path*. The unique heavy path originating at the root r is called the *main heavy path*. The set of all heavy paths is denoted as 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. Thus a tree may have multiple heavy path decompositions.

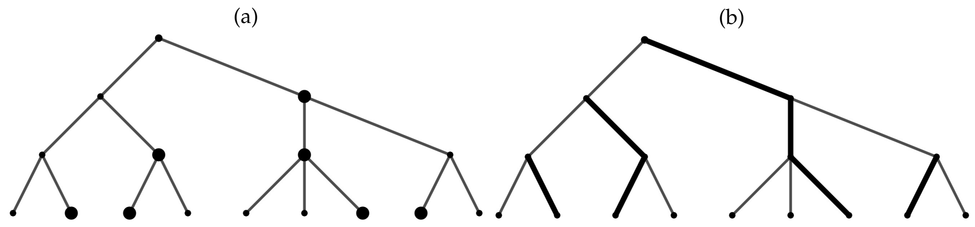


Figure 2.5: Example of a heavy path decomposition. Figure a) shows the heavy nodes, Figure b) the correspondig heavy paths.

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

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

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

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

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

Remark. A node $v \neq r \in V$ on the main heavy path of T is always a heavy node, per definition of a heavy path.

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 has 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. A set $C \subset X$ 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. In some literature a binary tree is defined as a tree, where every node has ≤ 2 children. This contains the possibility of nodes having exactly 1 child. In the context of this thesis, that possibility is not desired. In a full binary tree, all nodes are either a leaf or have exactly 2 children. This results in trees with a nice property:

Lemma 2.21. Let a full binary tree T with n leaves be given. Then T has $2n - 1$ nodes over all.

Proof. Proof by Induction. For $n \in \{0,1,2\}$ this is trivial. Assume the statement holds $\forall\, m \leq n$.
Let l be the left child and r the right child of T's root. Since T is a full binary tree, T_l and T_r also have to be full binary trees. The number of leaves in T_1 and T_r shall be denoted as m_l and m_r respectively.

$$\begin{aligned}
\Rightarrow\ m_l + m_r &= n \\
\Rightarrow\ |V(T_l)| = 2m_l - 1,\ |V(T_r)| &= 2m_r - 1 \\
\Rightarrow\ |V(T)| &= |V(T_l)| + |V(T_r)| + 1 \\
&= (2m_l - 1) + (2m_r - 1) + 1 \\
&= 2(m_l + m_r) - 1 = 2n - 1
\end{aligned}$$

□

Thus comparing two full binary trees with the same number of leaves implies that they have exactly the same number of nodes overall.

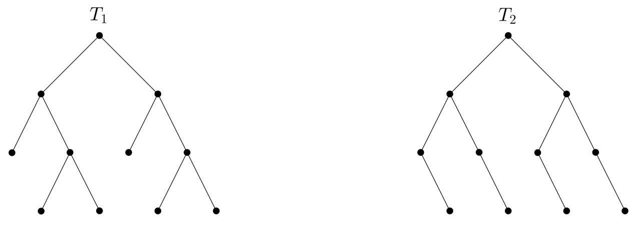


Figure 2.6: An illustration of two binary trees. In both trees all nodes have ≤ 2 children. However, only T_1 is a full binary tree because no node has exactly 1 child.

2.2 Other necessary Tools

Definition 2.22. 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. Let T' be the resulting tree, then:

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

.

Definition 2.23. Let $T = (V, E)$ be a rooted labelled tree and let o be one of the basic edit operations introduced above. The tree $o(T)$ is defined as the result of executing the operation o on tree T.
Furthermore let $o' = (o'_1, o'_2, ..., o'_n)$ be a finite sequence of basic edit operations. Applying these basic operations o'_i, $1 \leq 1 \leq n$ consecutively results in tree $o'(T)$:

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

Definition 2.24. 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

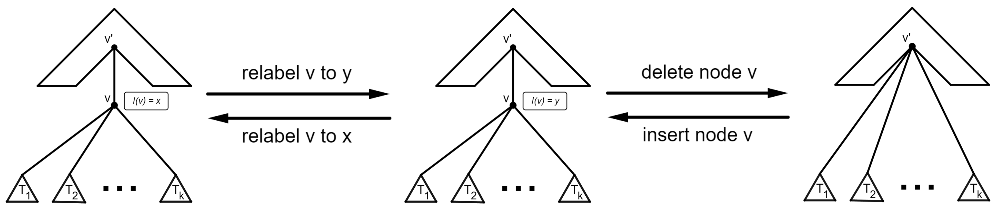


Figure 2.7: Illustration of the basic tree edit operations relabelling, deleting and inserting.

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. The costs of o' is defined as the sum of costs of the individual operations:

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

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

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 occurs in many counting problems. They are recursively defined as follows:

$$C_0 = 1$$
$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1} \text{ for } n \geq 1$$

Remark. The following two counting problems are examples where the Catalan numbers play a role:

- Counting the number of pairwise different expressions containing n pairs of parentheses where any prefix of the expression contains at least as many opening parentheses "(" as closing ones ")". [18]
- Counting the number of pairwise different full binary trees with $n+1$ leaves.

Lemma 2.27. *The number of pairwise different full binary trees with $n+1$ leaves is exactly the nth-Catalan number C_n.*

Proof. Proof by Induction. For $n = 0$ the statement is trivial: There is only 1 tree with exactly 1 leaf, which is a single node.
Suppose the inductive statement holds true for all $m < n$.
For $k, l \in \mathbb{Z}_{>0}$ define the following function:

$$C(k,l) := \left| \left\{ T \,\middle|\, \begin{array}{l} T \text{ is a full binary tree where the left subtree} \\ \text{has } k \text{ leaves and the right one has } l \end{array} \right\} \right|$$

For an integer $i \in \mathbb{Z}_{\geq 0}$, $i < n+1$, the number of pairwise different full binary trees with $n+1$ leaves, where the left subtree has i leaves is $C(i, n+1-i)$. Because of the inductive statement this number can be determined:

$$C(i, n+1-i) = C_{i-1}C_{n+1-i-1}$$

Summing up $C(i, n+1-i)$ over all possible i leads to the number of pairwise distinct full binary trees with $n+1$ leaves:

$$\begin{aligned} |\{\text{full binary trees with } n+1 \text{ leaves}\}| &= \sum_{i=1}^{n} C(i, n+1-i) \\ &= \sum_{i=1}^{n} C_{i-1}C_{n-(i-1)-1} = \sum_{i=0}^{n-1} C_i C_{n-i-1} = C_n \end{aligned}$$

□

3 Tree Edit Distance

In this chapter the so called *tree edit distance* is introduced and discussed. In addition to the historic background, multiple dynamic programming approaches to compute the tree edit distance between two trees are presented. This chapter is based on Demaine et al.'s paper [6] 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:

$$o^*(T_1) = o_n^*(...(o_1^*(T_1)) = T_2$$
$$c(o^*) = \sum_{i=1}^{n} c(o_i^*) = \min_{\substack{o \text{ sequence of} \\ \text{edit operations}}} \{c(o) \mid o(T_1) = T_2\}$$

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

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

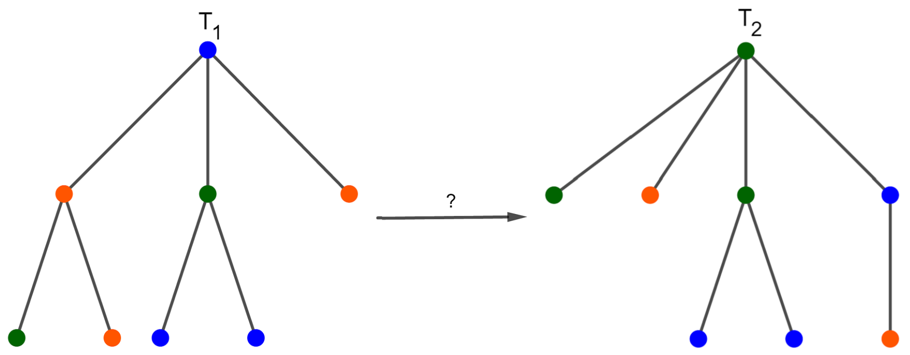
Definition 3.2. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two rooted ordered labelled forests and the rest as in the Definition 3.1. The tree edit distance $\delta(T_1, T_2)$ can be extended trivially to forests.

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. For the sake of illustration, the different labels are represented by different colours. 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 Figure 3.1 and the sequence of operations that performs the transformation from T_1 to T_2. Assuming that every operation costs the same then the sequence shown in the figure would be an optimal one.

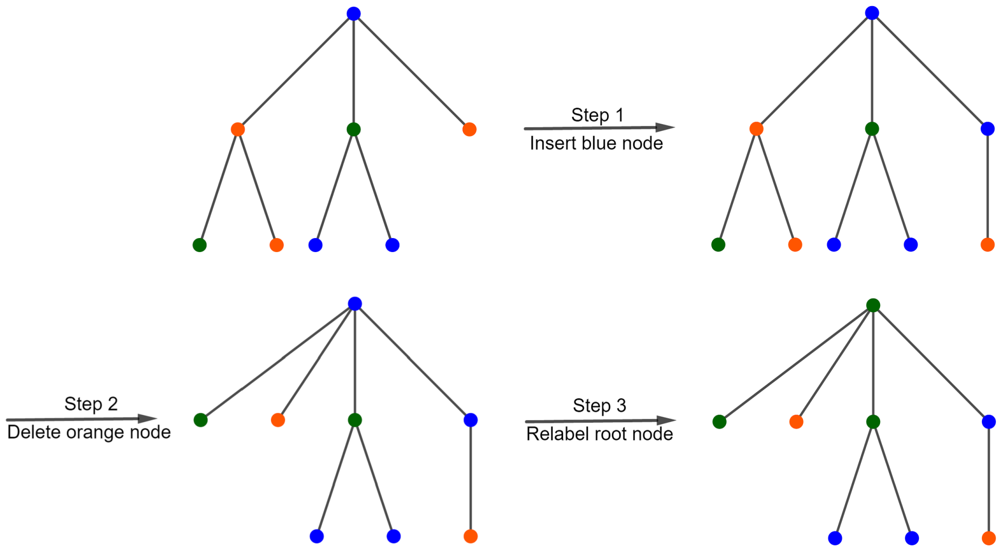
The tree edit distance is used in the fields of structured text databases, computer vision and in bioinformatics. In the last field one needs to compare the secondary structure of RNA-molecules without the disadvantages of other approaches. A more detailed description about this topic can be found in these papers: [10, 15, 22].

3.2 Short History of the Tree Edit Distance

The tree edit distance was introduced by Tai in the year 1979 [19]. 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 decade until Shasha and Zhang [17] came up with a dynamic programming approach that improved the running time to $O(t_1^2 t_2^2)$. Later on Klein [13] started with his algorithm, changed the branching strategy and was able to further improve the running time to $O(t_1 t_2^2 \log t_1)$. 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.



(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 there is an illustration of a 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.

Figure 3.1: The tree edit distance is a very intuitive way of comparing trees.

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. Some other algorithms can be found in the following papers: [1, 2, 20]

This thesis concentrates on the dynamic programming approaches of Shasha and Zhang, Klein and last but not least Demaine et al. However, their discussions will remain rather short.

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 cost functions for the basic editing operations be given. 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 $\delta(T_1, T_2)$, the problem of solving $\delta(T_1', T_2')$ is encountered. Define $\mathcal{R}_A$ as the set of all relevant subproblems.
A pairing (T_1', T_2') is called a *trivial subproblem* if and only if (T_1', T_2') is a relevant subproblem and $\exists i \in \{1, 2\}$ s.t. $T_i' = (\emptyset, \emptyset)$. Define $\mathcal{T}_A \subset \mathcal{R}_A$ 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. Because of the symmetry between deleting a node in T_1 and inserting a proper node in T_2 and vice versa, the set of basic edit operations can be restricted to relabelling and deleting.

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.

Any branching strategy only considers the two rightmost or the two leftmost roots of T_1 and T_2. Either one of them gets deleted or the two roots are matched. In the first case the resulting relevant subproblem contains exactly one node less than the original one. Matching two roots leads to two separated relevant subproblems:

Suppose matching 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. Thus matching r_{T_1} with r_{T_2} 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. Furthermore let the pair $(T_1', T_2') \in \mathcal{R}_A$ be a relevant subproblem.
The dynamic programming approach A associates the pairing (T_1', T_2') with the direction *left* or *right*. In the first case A branches at (T_1', T_2') by considering the two leftmost roots $l_{T_1'}$ and $l_{T_2'}$. Otherwise it operates on the two rightmost roots $r_{T_1'}$ and $r_{T_2'}$.
A mapping $S_A : \mathcal{R}_A \mapsto \{\text{left,right}\}$ is called 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 they are ordered according to the post order indexing. Consider the problem of computing $\delta(T_1, T_2)$ with dynamic programming approach A with the following decomposition strategy:*

$$S_A(T_1', T_2') = \textit{right}\ \forall (T_1', T_2') \in \mathcal{R}_A \setminus \mathcal{T}_A$$

Then:

$$\forall (T_1', T_2') \in \mathcal{R}_A \setminus \mathcal{T}_A : \exists\, i_1 \leq j_1,\, i_2 \leq j_2 \in \mathbb{N} \text{ s.t.}$$

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

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

Remark (Remark 2)*.* Let two induced subtrees $T_{v'}$ and $T_{v''}$ of T be given s.t. $V(T_{v'}) \cap V(T_{v''}) = \emptyset$ and 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. Proof by induction. For T_1 and T_2 the claim is trivially true. Assume that the claim holds for a relevant subproblem (T_1', T_2'). Therefore there exists some indexes i_1, j_1, i_2, j_2 as in the lemma. There are 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, if $i_1 = j_1$, $r_{T_1'}$ was 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 matching the roots splits 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. All nodes in R_{T_1} have a higher index than all the nodes in $T_1 - R_{T_1}$. according to Remark 2.
   $\Rightarrow \exists k_1$ 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 a trivial upper bound on the number 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. 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.

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

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

Running time: The running time is dependent on the number of relevant subproblems. However, there are upper bounds on the numbers of different subforests of T_1 and T_2 that appear in any relevant subproblem. Multiply those bounds together yields an overall upper bound on the number of relevant subproblems.
For the computation of these bounds keyroots and prefixes, as defined in the first chapter, play a critical role: 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 $i_{T_1'} = 1$, T_1' is a prefix of $T_1^\circ = (T_1)_{r_1}^\circ$ where r_1 is the root of T_1, assuming $j_{T_1'} < t_1$.

If $i_{T_1'} > 1$, 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.

Thus any subforest of T_1, which appears in a relevant subproblem, is a prefix of an induced subtree $(T_1)_v$ for some $v \in \text{keyroots}(T_1)$. Thus summing up all such prefixes will be an upper bound on the number of relevant subproblems:

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

Shasha and Zhang went on to prove the missing inequality:

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

Combining all the parts together leads to a running time of

$$O(\min\{t_{1,l}, t_{1,h}\} \min\{t_{2,l}, t_{2,h}\} t_1 t_2)$$

. □

3.3.2 Klein's algorithm

Klein [13] 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:

$$S_K(T_1', T_2') = \begin{cases} \text{left} & |V(L_{T_1'})| \le |V(R_{T_1'})| \\ \text{right} & \text{otherwise} \end{cases}$$

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 the most efficient decomposition strategy. They proved a running time of $O(t_2^2 t_1(1 + \log \frac{t_1}{t_2}))$ and lastly showed that this is also 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 length. All proofs and further details can be found in 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 3.2. As the caption explains the number at each node shall demonstrate the order among children. Looking 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. During the computation of $\delta((T_1)_v, T_2)$ with $v \in TopLight(T_1)$ all pairs $((T_1)_u^\circ, (T_2)_w^\circ)$, where*

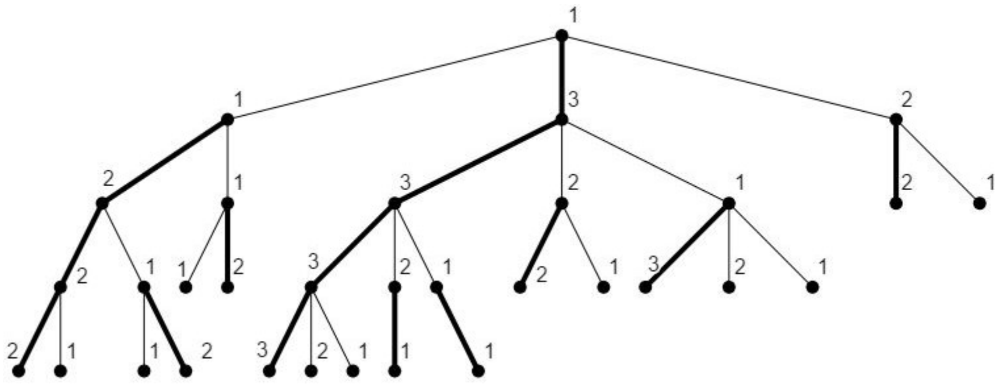


Figure 3.2: The number at each node indicates the order among siblings in which they are considered according to S_D.

$u \in T_1, w \in T_2$ and both not on the respective main heavy paths of T_1, T_2, are encountered as relevant suproblems and therefore the corresponding distances $\delta((T_1)^\circ_u, (T_2)^\circ_w)$ are computed as well.

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

Theorem 3.10. *The distance $\delta(T_1, T_2)$ gets computed recursively as follows:*

1. *If $|V(T_1)| < |V(T_2)|$ 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, the distance $\delta(T_1, T_2)$ can be computed in $O(t_2^2 t_1(1 + \log \frac{t_1}{t_2}))$ time.

3.3.4 Lower bound on decomposition algorithms

The optimal running time for decomposition algorithms has a lower bound which has already been achieved by the previous algorithm of Demaine et al. This lower bound was proven by Demaine et al. using specifically structured trees illustrated in Figure 3.4. The necessary ideas to prove the tight lower bound are similar to the ones used in the proof of following lemma:

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 respectively, such that the number of relevant subproblems is $\Omega(t_2^2 t_1)$*

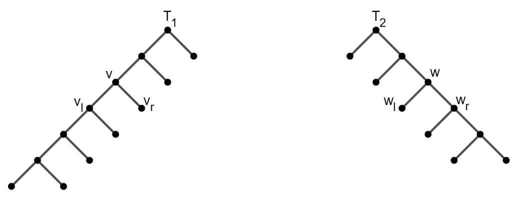


Figure 3.3: Sketch of T_1 and T_2 which fulfilla lower bound on the running time for each decomposition algorithm of $\Omega(t_2^2 t_1)$

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 $v \in T_1, w \in T_2$ is a relevant subproblem for S. The number of such relevant subproblems, where v and w are inner nodes of T_1 and T_2, can be counted: For an inner node x, let x_l denote the left child of x and x_r the right child of x. In the forest $(T_1)_v^\circ$ the rightmost root is v_r and in $(T_2)_w^\circ$ the leftmost root is w_l. In each step S decides the direction from which side a node has to be deleted. However if the strategy chooses left, the deletion shall be performed on T_1, otherwise on 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 $\min\{|(T_1)_v^\circ|, |(T_2)_w^\circ|\}$ steps until every relevant subproblem of $((T_1)_v^\circ, (T_2)_w^\circ)$ is found. Since v_r and w_l are the outermost roots, the computational paths of $((T_1)_v^\circ, (T_2)_w^\circ)$ and $((T_1)_{v'}^\circ, (T_2)_{w'}^\circ)$ are completely disjoint. Because of their structure there are approximately $\frac{t_1}{2}$ and $\frac{t_1}{2}$ internal nodes in T_1 and T_2 respectively, yielding

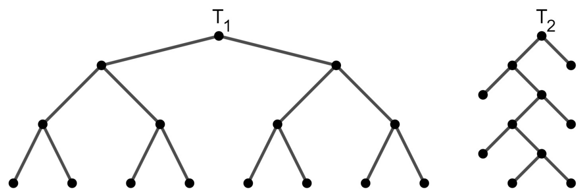


Figure 3.4: T_1 and T_2 used to prove a lower bound of $\Omega(t_2^2 t_1(1 + \log \frac{t_1}{t_2}))$

the following equation:

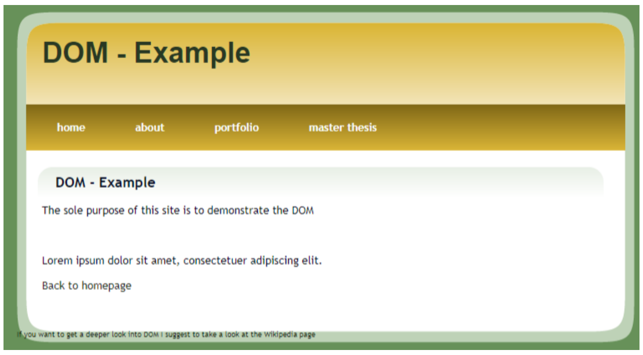
$$\sum_{(v,w)\text{internal nodes}} \min\{|(T_1)_v^\circ|, |(T_2)_w^\circ|\} = \sum_{i=1}^{\frac{t_1}{2}} \sum_{j=1}^{\frac{t_2}{2}} \min\{2i, 2j\} = \Omega(t_2^2 t_1).$$

□

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

The strict requirement regarding a trees hierarchy is one problem of the standard tree edit distance. The hierarchy refers to the fixed parent-to-child relationship within an ordered tree. Another inconvenience is the fixed ordering among siblings. If a node $v \in T_1$ gets mapped to a node $w \in T_2$ during the computation of the tree edit distance, the children of v have to get mapped to descendants of w or get deleted. In some domains, the most representative matching may not fulfil these requirements. One of these domains is the DOM (Document Object Model) of a website. A standard HTML-based webpage can easily be structured according to the respective HTML-tags. Take a look at the website in Figure 4.1 and its HTML-code in Listing 4.1.


DOM - Example

home about portfolio master thesis

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">
```

```html
    <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 the complete DOM-tree illustrated in Figure 4.2.

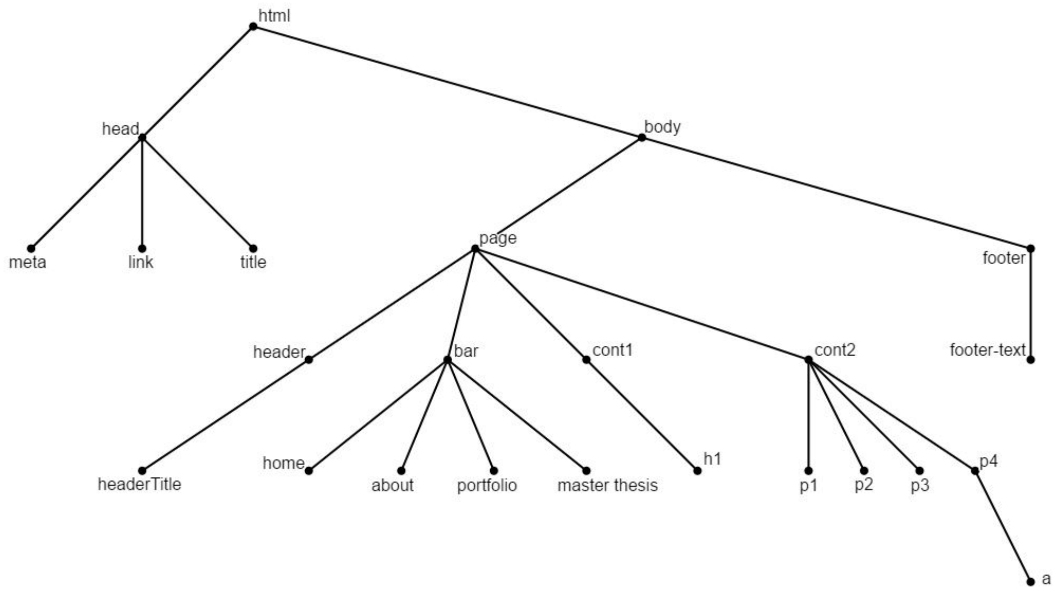
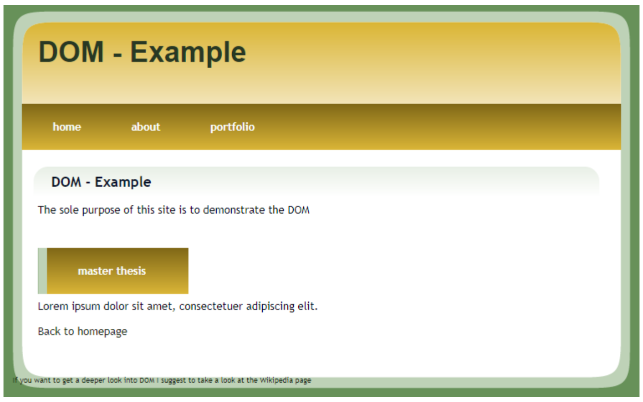


Figure 4.2: Complete DOM-tree of the example website.

Intuitively, any small change to the website should not lead to a big distance between the corresponding 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. However, deleting the whole header menu and therefore reducing the website's functionality leads to a smaller tree edit distance.

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. [14] developed an algorithm that matches nodes with similar labels and penalizes edges that break up sibling groups or violate the



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.

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 into another subtree. A violation of this kind needs to have some 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

Definition 4.1. Let T_1 and T_2 be two rooted ordered labelled trees with a set of labels Σ. Define G_{T_1,T_2} to be the complete bipartite graph on the following set of nodes:

$$G := (\{V(T_1) \cup \otimes_1\} \dot\cup \{V(T_2) \cup \otimes_2\}, E(G_{T_1,T_2}))$$

Hence the edge set $E(G_{T_1,T_2})$ is defined as the set:

$$E(G_{T_1,T_2}) := \{\{v_1, v_2\} \mid v_i \in \{V(T_i) \cup \otimes_i\} \text{ for } i = 1,2\}.$$

The nodes $\otimes_i$ are so called *no-match* nodes.

Remark. Every edge $e = \{v_1, v_2\} \in E(G_{T_1,T_2})$ represents matching a node v_1 to a node v_2. If $v_2 = \otimes_2$, the edge e represents 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 $\{\otimes_1, v_2\}$.

Definition 4.2. Let T_1 and T_2 be two rooted ordered labelled trees with a set of labels Σ and the graph G_{T_1,T_2} be given. A set of edges $M \subseteq E(G_{T_1,T_2})$ is called a *flexible matching*, if the following statements are true:

1. $\forall v_1 \in V(T_1) : \exists! v_2 \in \{V(T_2) \cup \otimes_2\}$ s.t.: $\quad (v_1, v_2) \in M$
2. $\forall v_2 \in V(T_2) : \exists! v_1 \in \{V(T_1) \cup \otimes_1\}$ s.t.: $\quad (v_1, v_2) \in M$

The set of all flexible matchings is denoted as M_{T_1,T_2}

Remark. Note that the no-match nodes do not have any restrictions on them. A no-match node may be a part of 0 edges or it may be arbitrarily often matched.

Every edge $e \in E(G_{T_1,T_2})$, $e \in V(T_1) \times V(T_2)$ is assigned some cost function $c(e)$:

$$c(e) = c_r(e) + c_a(e) + c_s(e). \tag{4.1}$$

The exact definitions follow later. In short terms, these three summands represent the costs that were 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. All the other edges, namely the ones connecting tree nodes with no-match nodes, have a fixed constant cost w_n, only depending on the number of nodes in the trees.

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 $M(v) \neq \otimes_2$. Recall that $C(v) \subset T_1$ is defined as the set of children of v, $P(v) \in T_1$ as the parent of v and $S(v) \subset T_1$ as the sibling group of v. The set $V(v)$ contains all children of v that violate the ancestry condition, i.e.:

$$V(v) := \{v' \in C(v) \mid M(v') \in T_2 \setminus C(M(v))\} \tag{4.2}$$

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:

$$c_a(\{v,w\}) := \omega_a(|V(v)| + |V(w)|) \tag{4.3}$$

The costs $c_s(e)$ are dependent on the *sibling-invariant subset* $I_M(v)$ of v. This set $I_M(v)$ is defined as the set of all siblings of v which are mapped into the same sibling group as v:

$$I_M(v) := \{v' \in S(v) \mid M(v') \in S(M(v))\} \tag{4.4}$$

Accordingly the *sibling-divergent subset* $D_M(v)$ of v contains all siblings of v

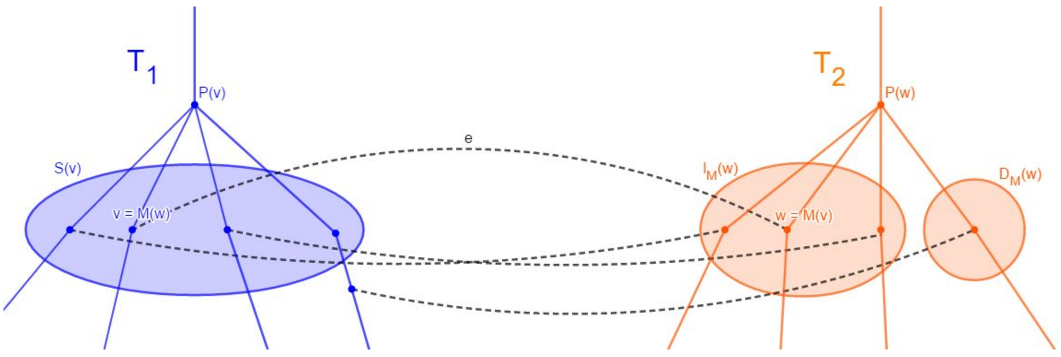


Figure 4.4: A visual representation of the above described tree concepts.

which are mapped to a node in $T_2 \setminus S(M(v))$:

$$D_M(v) := \{v' \in S(v) \mid M(v') \in T_2 \setminus S(M(v))\} \tag{4.5}$$

$$= \{v' \in S(v) \setminus I_M(v) \mid M(v') \neq \otimes_2\} \tag{4.6}$$

Finally, the set of *distinct sibling families* is defined as the union of all sibling groups, that the siblings of v map into:

$$F_M(v) = \bigcup_{v' \in S(v)} P(M(v')) \tag{4.7}$$

Having defined all these concepts, the costs for sibling group violations, depending on a constant ω_s, can be formulated as follows:

$$c_s(\{v,w\},M) := \omega_s\left(\frac{|D_M(v)|}{|I_M(v)||F_M(v)|} + \frac{|D_M(w)|}{|I_M(w)||F_M(w)|}\right) \tag{4.8}$$

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.

Definition 4.3. Let T_1, T_2 two rooted ordered trees, the corresponding graph G_{T_1,T_2} as well as 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 which fulfils the following equation:

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

where $c(e)$ is defined as described in Equations (4.1), (4.3), (4.8). Then $c(M^*)$ is called the *flexible tree edit distance*.

4.2 Approximation and Conclusion

As mentioned in the introduction of this chapter, computing the flexible tree edit distance is an $\mathcal{NP}$-hard task. This statement can be proven by reducing the 3-partition problem to finding an optimal flexible tree matching. Once again all details can be found in Kumar et al.'s paper [14]. Hence, there exists no efficient algorithm that computes the flexible tree matching to optimality, unless $\mathcal{P} = \mathcal{NP}$. But there are stochastic optimization algorithms to get an approximation of the flexible tree edit distance. Kumar et al. 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 having more fixed edges leads to more accurate bounds. Furthermore the authors give advices on how to adapt the cost factors ω_r, ω_s, ω_a and ω_n step by step in order 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. Diverse applications may value sibling group or ancestry violations differently and their significance can be modelled by appropriately chosen values for coefficients ω_r and ω_s. Thus the cost model for the flexible tree matching can be tuned to reflect the real life problem as accurately as possible. Having a database of exemplar matchings opens up the possibility to implement a learning cost model which improves the coefficients to get the better results.
Nevertheless, there can not be an efficient algorithm to calculate the flexible tree edit distance. Therefore all results are more suggestive than definitive, just like the problem itself.

5 Robinson Foulds Metric

A phylogenetic tree or evolutionary tree is a rooted branching diagram used in the field of biology, which shows the evolutionary connectedness of different species. Every species is represented by a node and is connected to its immediate evolutionary ancestor. A leaf is called a taxon and is labelled with the species it represents. All in all, there is 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 labels of its leaves are called taxons and represent the investigated species. All interior nodes stand for a common ancestor of the taxons present in its induced subtree. Therefore the root of a phylogenetic tree is the most recent common ancestor of all 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

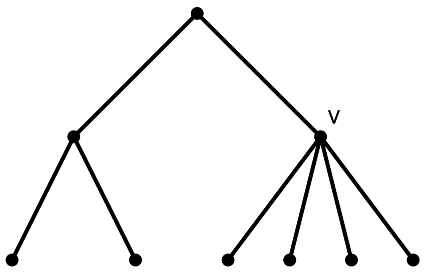


Figure 5.1: A rooted phylogenetic tree. The node v is a multifurcation since its degree is larger than 3. The ancestry relations aren't fully resolved for the children of v.

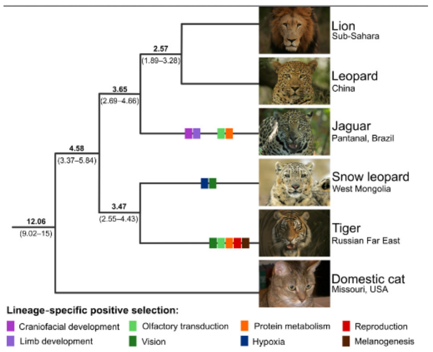


Figure 5.2: A simplified phylogenetic tree that shows the evolutionary connection between five big cats and the domestic house cat. The complexity of finding this task is described in the paper by Figueiró et al. [8].

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 for phylogenetic trees was introduced by Robinson and Foulds [16]. They defined the term *clade* describing a group of leaves that have a common ancestor they do not share with any other leaf.

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. The Robinson-Foulds metric d_{RF} is defined as follows:

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

Remark. Assuming that both trees T_1,T_2 have the same set of taxa X with $n := |X|$ implies that the number of interior nodes in both trees is bounded by $n-1$. Since the trivial clade X is induced by the tree's root, the inequality $d_{RF}(T_1, T_2) \leq n-2$ holds.

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. An example of this behaviour is illustrated in Figure 5.3. 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 every clade in T_1 contains both nodes 1 and 2, whereas in tree T_2 every clade either contains node 1 or 2, but never both. This implies that

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

However the third tree T_3 also has the same Robinson Foulds distance to both trees T_1 and T_2. This points to another big disadvantage of the Robinson Foulds distance. The distribution of 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 metric does not compare the structure of trees. Otherwise the distance between T_1 and T_2 would be much smaller than the one to tree T_3.

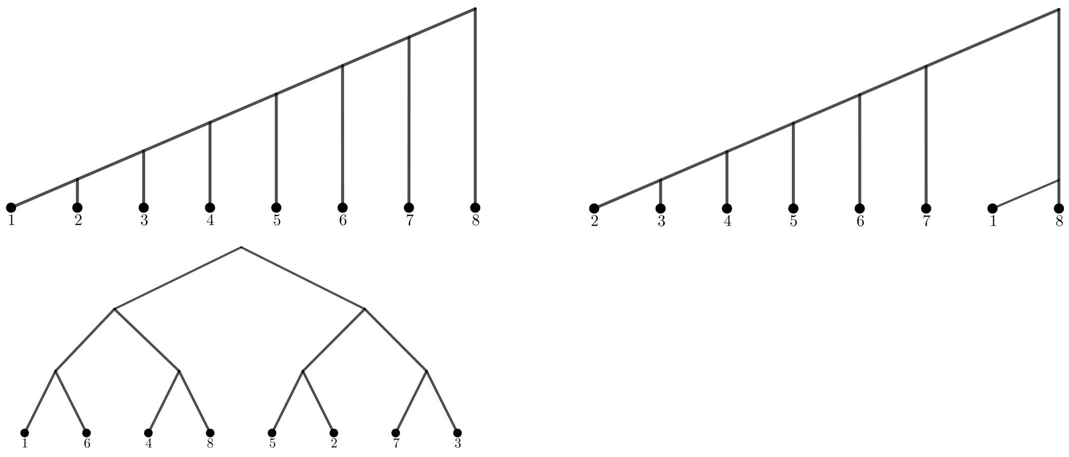


Figure 5.3: Three trees having the maximal RF-distance on the set of taxa $\{1, ..., 8\}$

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 $\underline{G}_{T_1,T_2}$ 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. Define $\underline{G}_{T_1,T_2}$ as the complete bipartite graph on the following set of nodes:

$$\underline{G}_{T_1,T_2} : \mathcal{C}^*(T_1) \times \mathcal{C}^*(T_2)$$

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. A mapping δ is called a *cost function* if it possesses the following properties:

- $\delta : \mathcal{P}(X) \times \mathcal{P}(X) \mapsto \mathcal{R}_{\geq 0} \cup \{\infty\}$, $\mathcal{P}(X)$ being 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 between 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 $\underline{G}_{T_1,T_2}$ 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 $\underline{G}_{T_1,T_2}$. A clade $C \in \mathcal{C}^*(T_1)$ is called *unmatched*, if $\nexists C' \in \mathcal{C}^*(T_2)$ s.t. $(C, C') \in M$, analogously for clades of T_2. Furthermore the *costs of a matching* M, $d_\delta(M)$, is defined 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 $\underline{G}_{T_1,T_2}$ yields $\overline{d}_\delta(T_1, T_2)$:

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

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.:

$$\overline{d}_{\delta_{RF}}(T_1, T_2) = \min_{M \text{ matching in } \underline{G}_{T_1,T_2}} d_{\delta_{RF}}(M) = d_{\delta_{RF}}(M^*)$$

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:

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

$\square$

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 $\underline{G}_{T_1,T_2}$ and a cost function δ be given as defined in Definitions 5.2 and 5.3. The task of computing a matching M^* satisfying $\overline{d}_\delta(T_1, T_2) = d_\delta(M^*)$ can be simplified by the following model:*
For any edge $(C, C') \in E(\underline{G}_{T_1,T_2})$ 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 of clades is equivalent to finding a maximal cost matching 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}. The following implications are trivial:

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

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

Remark (Example). Compute the distance defined in Definition 5.3 between trees T_1, T_2 from Figure 5.3 with respect to the size of the symmetric difference between sets:

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

Using Lemma 5.6 provides an easy way to compute the requested distance. Figure 5.4 illustrates the graph $\underline{G}_{T_1,T_2}$ with the corresponding edge weights. The optimal matching in $\underline{G}_{T_1,T_2}$ can be seen very easily:
Let M^* be the matching represented by the bold edges in Figure 5.4. For every edge $(C_1, C_2) \in M^*$, the weight $\omega(C_1, C_2)$ is maximal among all edges starting at C_1, implying optimality.

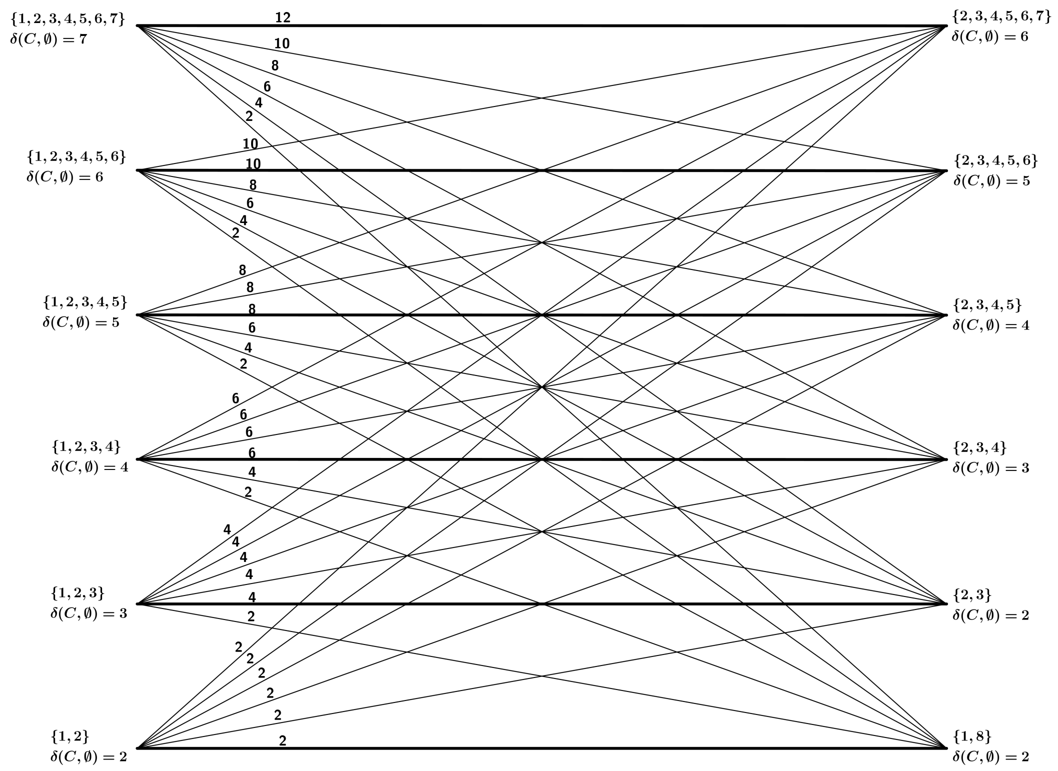

$\{1,2,3,4,5,6,7\}$ $\delta(C,\emptyset)=7$
$\{1,2,3,4,5,6\}$ $\delta(C,\emptyset)=6$
$\{1,2,3,4,5\}$ $\delta(C,\emptyset)=5$
$\{1,2,3,4\}$ $\delta(C,\emptyset)=4$
$\{1,2,3\}$ $\delta(C,\emptyset)=3$
$\{1,2\}$ $\delta(C,\emptyset)=2$
$\{2,3,4,5,6,7\}$ $\delta(C,\emptyset)=6$
$\{2,3,4,5,6\}$ $\delta(C,\emptyset)=5$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

Figure 5.4: Illustration of graph $\underline{G}_{T_1,T_2}$. On the left hand side the nodes correspond to non-trivial clades of T_1, on the right hand side they represent the ones of T_2. The small number on top of an edge is the value of its weight. The maximum cost matching M^* is highlighted by the bold edges.

Analogously the other distances can be computed as well:

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

This distance measure apparently delivers values which are closer to an intuitive answer. The distances indicate significant similarities between trees T_1 and T_2. Furthermore they point out that T_3 has a different structure than the other trees.

Of course such values can only be compared to distances with respect to the same cost function δ. Generally, comparing distances $\bar{d}_\delta(T_1, T_2)$ and $\bar{d}_{\delta'}(T_1, T_2)$ is not feasible without adding further context.
Comparing $\bar{d}_{\delta_{RF}}(T_1, T_2) = 6 < 8 = \bar{d}_{\delta_{sym}}(T_1, T_2)$ doesn't yield any information. However one could get data from following 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:

$$\{1,2\} \in \mathcal{C}^*(T_1) \subset \{1, ..., j\} \in \mathcal{C}^*(T_1) \forall j \geq 3,$$

however an analogous statement doesn't hold for its matched clade $\{1, 8\}$. Such behaviour can be avoided by demanding matchings to be arboreal:

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. Define $d_\delta(T_1, T_2)$ as follows:

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

The value $d_\delta(T_1, T_2)$ is called 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 the generalized Robinson-Foulds distance will be abbreviated by **gRF**.
2. The gRF $d_\delta(T_1, T_2)$ and the previous distance measure $\overline{d}_\delta(T_1, T_2)$ are conceptually very similar. The only difference is, that the gRF minimizes only over arboreal matchings.
3. Trivially the inequality $\overline{d}_\delta(T_1, T_2) \leq d_\delta(T_1, T_2)$ holds.
4. The statement in Lemma 5.6 is true for $d_\delta(T_1, T_2)$ as well. Its proof works the same way, but instead of maximizing over all matchings, 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:

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

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 $k \to \infty$:*

$$\delta_k(C_1, C_2) \xrightarrow[k\to\infty]{} \begin{cases} 0 & \textit{if } C_1 = C_2 \\ 1 & \textit{if } C_1 \neq C_2 \end{cases}$$

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: $C_1 \neq C_2$

$$\begin{aligned} &\Rightarrow (C_1 \cup C_2) \setminus (C_1 \cap C_2) \neq \emptyset \\ &\Rightarrow |C_1 \cup C_2| > |C_1 \cap C_2| \\ &\Rightarrow \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} < 1 \\ &\Rightarrow (\frac{|C_1 \cap C_2|}{|C_1 \cup C_2|})^k \xrightarrow[k\to\infty]{} 0 \\ &\Rightarrow \delta_k(C_1, C_2) = 1 - (\frac{|C_1 \cap C_2|}{|C_1 \cup C_2|})^k \xrightarrow[k\to\infty]{} 1 \end{aligned}$$

□

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 gRF metric introduced by δ_k as defined in Equation (5.2) is called Jaccard-Robinson-Foulds metric (**JRF**) of order k and is denoted by $d^{(k)}_{JRF}(T_1, T_2)$.

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 threshold, the original problem of finding a correct assignment for ϕ has a solution. All details can be found in the original paper [3].

Theorem 5.12. *For an instance of the arboreal matching using the symmetric difference as cost function δ and an integer k, it is NP-complete to decide whether there exists an arboreal matching of cost at most k.*

5.3.3 An Integer Linear Program

One way to approach the $\mathcal{NP}$-complete problem above 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. Suppose the clades in $\mathcal{C}(T_i)$ are numbered from 1 to $|V_i|$ for $i \in \{1,2\}$. Introduce $x_{i,j}$ as a binary indicator variable representing whether the i-th clade C_i in $\mathcal{C}(T_1)$ is matched with the j-th clade C'_j in $\mathcal{C}(T_2)$. The edge weights introduced in Equation (5.1) ensure that finding a maximum cost assignment on the indicator variables leads to a minimum cost matching. To recapitulate, the value for $\omega(C_i, C'_j)$ is:

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

The constraints on an assignment of indicator variables have to ensure, that the result is an arboreal matching. Therefore the set $\mathcal{I}$ is introduced with the following properties:

$$\mathcal{I} := \left\{ \{(i,j),(k,l)\} \,\middle|\, \begin{array}{l} \text{The edges } (C_i, C'_j) \text{ and } (C_k, C'_l) \text{ violate the conditions} \\ \text{for arboreal matchings defined in Definition 5.7} \end{array} \right\} \tag{5.3}$$

Theorem 5.13. *Let T_1 and T_2, two rooted phylogenetic trees on the same set of taxa X with $|X| =: n$, and a cost function δ be given. Furthermore a fixed order of all non-trivial clades in $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$ and the cost function $\omega(C_i, C'_j)$ shall be given. Let $x_{i,j}$ be the indicator variables of the following integer linear problem (ILP):*

$$\max \sum_{i=1}^{n-2} \sum_{j=1}^{n-2} \omega(C_i, C'_j) x_{i,j} \tag{5.4}$$

$$\text{s.t.} \sum_{j=1}^{n-2} x_{i,j} \leq 1 \qquad \forall i \in \{1, ..., n-2\} \tag{5.5}$$

$$\sum_{i=1}^{n-2} x_{i,j} \leq 1 \qquad \forall j \in \{1, ..., n-2\} \tag{5.6}$$

$$x_{i,j} + x_{k,l} \leq 1 \qquad \forall \{(i,j),(k,l)\} \in \mathcal{I} \tag{5.7}$$

$$x_{i,j} \in \{0,1\} \qquad \forall i \in \{1, ..., n-2\}, \forall j \in \{1, ..., n-2\} \tag{5.8}$$

Suppose an optimal assignment of the ILP is given by $(x^*_{i,j})$. 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. First and foremost, the requirement of binary decision variables in Constraints (5.8) ensure that every edge is either chosen or not. Furthermore

Constraints (5.5) and (5.6) restrict the chosen set of edges to be a, not necessarily complete, matching. Last but not least, because of Constraints (5.7) the resulting matching is arboreal, since $\mathcal{I}$ was constructed that way. As proven in Lemma 5.6, the arboreal matching of maximum cost with respect to the cost function (5.4) is an arboreal matching that minimizes the gRF $d_\delta(T_1, T_2)$. □

Remark. The number of non-trivial clades in a phylogenetic tree with the set of taxa X, $|X| = n$ is $n - 2$. Thus the running indices in the sums of inequalities (5.4), (5.5), (5.6) have an upper bound of $n - 2$.

The number of the ILP's constraints influences the running time of any algorithm that computes or approximates an optimal solution. It is obvious, that there are $2n - 4 = \mathcal{O}(n)$ restrictions handling the issue of ending up with a viable matching (5.5), (5.6).
Computing the number of restrictions for ensuring the matching to be arboreal, Inequalities (5.7), depends on the set $\mathcal{I}$. The following lemma shows the order of its size.

Lemma 5.14. *Let two phylogenetic trees T_1, T_2 with set of taxa X, $|X| =: n$, and the set $\mathcal{I}$ be given as described in Equation (5.3). Then the size of $\mathcal{I}$ is of order:*

$$|\mathcal{I}| = \Omega(n^2 \log(n)^2)$$

Proof. For a non-trivial clade $C^{(i)} \in \mathcal{C}(T_i)$, $i \in \{1, 2\}$ define the set of *non-trivial predecessor clades* as follows:

$$P(C^{(i)}) := \{C | C \in \mathcal{C}^*(T_i), C^{(i)} \subsetneq C\}$$

By definition every clade in $\mathcal{C}^*(T_i)$ corresponds to a node $v^{(i)}_{C^{(i)}} \in V(T_i)$. Thus every clade in the set of non-trivial predecessor clades $P(C^{(i)})$ corresponds to an inner node on the path from $v^{(i)}_{C^{(i)}}$ to the root of T_i.

Claim 1. Let $C^{(i)} \in \mathcal{C}(T_i)$, $i \in \{1,2\}$ be given, where both of them have a non-trivial predecessor: $|P(C^{(i)})| \geq 1$. Furthermore let $C^{(i)'} \in P(C^{(i)})$ be given for $i \in \{1,2\}$. Then the following inclusion holds:

$$\{(C^{(1)}, C^{(2)'}), (C^{(1)'}, C^{(2)})\} \in \mathcal{I}$$

Proof of Claim 1. The non-trivial predecessors are constructed in such a way, that $C^{(1)} \subsetneq C^{(1)'}$ and $C^{(2)} \subsetneq C^{(2)'}$. Thus the set of edges does not fulfil the conditions of Definition 5.7.

Counting the number of such pairings leads to a lower bound on the size of $\mathcal{I}$. For an inner node $v_i \in V(T_i)$ corresponding to a clade $C^{(i)}$ define the following value:

$$p(v_i) := |P(C^{(i)})|$$

This value suggests how many pairings of clades within tree T_i exist, where $C^{(i)}$ is the smaller clade. Let $P(T)$ be defined as follows:

$$P(T_i) := \{p(v) | v \in V(T_i), v \text{ inner node of } T_i\}$$

Claim 2. Let T_1, T_2 be two full binary trees with the same number of leaves. Let T_1 be *more balanced* than T_2, meaning:

$$\sum_{v \in V(T_1)} \text{height}(v) < \sum_{v \in V(T_2)} \text{height}(v)$$

Then the following statement is true:

$$P(T_1) < P(T_2)$$

Proof of Claim 2. If T_1 is more balanced, the same holds true for the trees induced by all inner nodes of T_1 and T_2 respectively. For an inner node v_i in T_i, the only difference between $\text{height}(v_i)$ and the value $p(v_i)$ is, that the height counts the root of T_i as well. Since the number of inner nodes are the same in both trees the following inequalities hold:

$$\begin{aligned}
\sum_{v \in V(T_1)} \text{height}(v) &< \sum_{v \in V(T_2)} \text{height}(v) \\
\Rightarrow \sum_{v \text{ inner node of } T_1} \text{height}(v) &< \sum_{v \text{ inner node of } T_2} \text{height}(v) \\
\Rightarrow \sum_{\substack{v \text{ inner node of } T_1 \\ v \neq r_{T_1}}} \text{height}(v) + \text{height}(r_{T_1}) &< \sum_{\substack{v \text{ inner node of } T_2 \\ v \neq r_{T_2}}} \text{height}(v) + \text{height}(r_{T_2}) \\
\Rightarrow \sum_{\substack{v \text{ inner node of } T_1 \\ v \neq r_{T_1}}} (p(v) + 1) &< \sum_{\substack{v \text{ inner node of } T_2 \\ v \neq r_{T_2}}} (p(v) + 1) \\
\Rightarrow \sum_{\substack{v \text{ inner node of } T_1 \\ v \neq r_{T_1}}} p(v) \quad + (n-2) &< \sum_{\substack{v \text{ inner node of } T_2 \\ v \neq r_{T_2}}} p(v) \quad + (n-2) \\
\Rightarrow P(T_1) &< P(T_2)
\end{aligned}$$

Let $f : \mathbb{N} \to \mathbb{N}$ be a function s.t. $f(n)$ is a tight lower bound on $P(T)$ for any full binary tree T with n leaves.

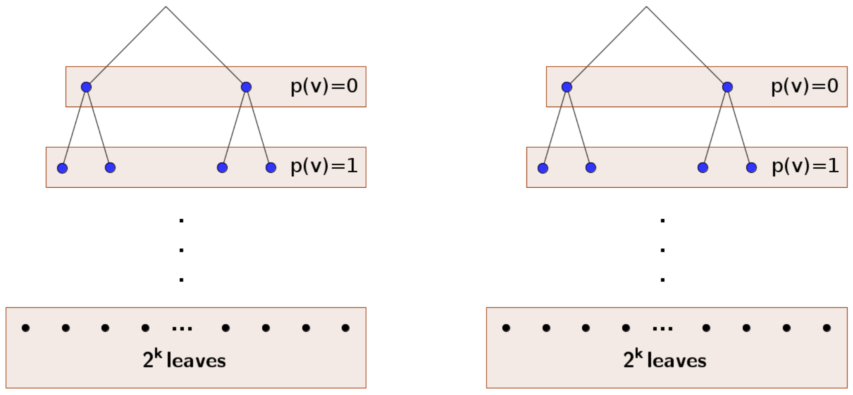
Claim 3. For $k \in \mathbb{N}_{\geq 2}$ the following equality holds:

$$f(2^k) = (k-3)2^k + 4$$

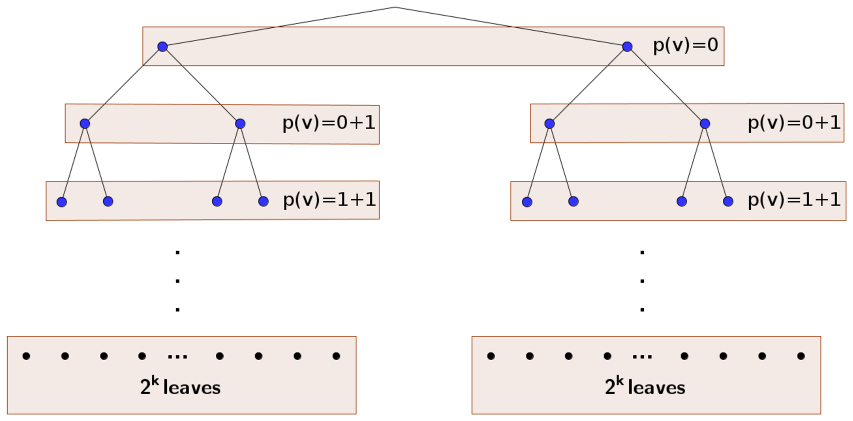
Proof of Claim 3. Proof by induction. Start with $k = 2$:
Let T be the perfectly balanced tree with 4 leaves. As shown in Claim 2, this tree has the smallest value $P(T)$ among all full binary trees with 4 leaves. This argument is supported by the fact that $P(T) = 0$.

$$f(2^2) = (2-3)2^2 + 4 = -1 * 4 - 4 = 0$$

For the induction step assume that the statement is true $\forall k < K$.
Let T' be the balanced tree with $2^{(K-1)}$ leaves and T the one with 2^K leaves. Constructing T can be done by taking two trees with the structure of T',



(a) Start with two perfectly balanced trees with 2^k leaves each.



(b) Connect them by adding a new root node and appending the roots of both trees as children.

Figure 5.5: The value of $p(v)$ increases by 1 for all inner nodes of the starting trees.

adding a root r_T and adding the roots of the smaller trees T' as children of r_T. This procedure is sketched in Figure 5.5 and simplifies the computation of $P(T)$:
Let $r_{T'}$ be a root of the left or right subtree of T. Since they are equivalent, the following argument works for both subtrees. The node $r_{T'}$ suddenly becomes an inner node in T. However $p(r_{T'}) = 0$ since the parent of $r_{T'}$ is the root of T which isn't a non-trivial predecessor. On the other hand, $p(v)$ increases by 1 for all inner nodes $v \in T'$ since they get a new non-trivial predecessor. Thus the overall sum within T' increases by the number of inner nodes in T', which is $2^{(K-1)} - 2$. This implies the following equality:

$$\begin{aligned}
\Rightarrow f(2^K) &= 2(f(2^{(K-1)}) + 2^{(K-1)} - 2) \\
&= 2(((K-1)-3)2^{(K-1)} + 4) + 2^K - 4 \\
&= (K-4)2^K + 8 + 2^K - 4 \\
&= (K-3)2^K + 4
\end{aligned}$$

Claim 4. Let $k, l \in \mathbb{N}_{\geq 2}$, s.t. $2^k > l$. Then:

$$f(2^k + l) = (k-3)2^k + 4 + (k-1)l$$

Proof of Claim 4. The variables k and l are chosen this way to find a value for $f(n)$ for all $n \in \mathbb{N}$. Claim 2 suggests that in order to minimize $f(n)$, one has to find a tree with n leaves which is as balanced as possible. Since $2^k > l$, it is clear that $2^k + l < 2^{(k+1)}$. Trivially the most balanced tree can be constructed the following way:

1. Start with a balanced tree T with 2^k leaves.
2. Choose l leaves of T and exchange them with full binary trees with 2 leaves.

Thus the newly constructed tree T' ends up with $2^k + l$ leaves. Altogether T' has $2l$ leaves with height $k+1$, the others stay at the height of k. Moreover T' has l more inner nodes than T. All of them have a height of k, so $p(v) = k - 1$

for all such nodes v. Therefore the following statement is true:

$$\begin{aligned} P(T') &= P(T) + (k-1)l \\ \Rightarrow f(2^k + l) &= f(2^k) + (k-1)l = (k-3)2^k + 4 + (k-1)l \end{aligned}$$

Claim 5. Let $n \in \mathbb{N}_{\geq 2}$. Then:

$$f(n) = n(\lfloor \log_2(n) \rfloor - 1) - 4 * 2^{\lfloor \log_2(n) \rfloor} + 4$$

Proof of Claim 5. Let $k, l \in \mathbb{N}$ s.t. $2^k > l$ and $n = 2^k + l$.

$$\begin{aligned} k = \lfloor \log_2(n) \rfloor, \quad l &= n - 2^k = n - \lfloor \log_2(n) \rfloor \\ \Rightarrow f(n) = f(2^k + l) &= (k-3)2^k + 4 + (k-1)l \\ &= (\lfloor \log_2(n) \rfloor - 3)2^{\lfloor \log_2(n) \rfloor} + (\lfloor \log_2(n) \rfloor - 1)(n - 2^{\lfloor \log_2(n) \rfloor}) + 4 \\ &= n(\lfloor \log_2(n) \rfloor - 1) - 4 * 2^{\lfloor \log_2(n) \rfloor} + 4 \end{aligned}$$

The order of $f(n)$ can be obtained easily:

$$\begin{aligned} f(n) &= \Theta(n(\lfloor \log_2(n) \rfloor - 1) + \Theta(4 * 2^{\lfloor \log_2(n) \rfloor}) + \Theta(4) \\ &= \Theta(n \log(n)) + \Theta(n) + \Theta(1) \\ &= \Theta(n \log(n)) \end{aligned}$$

The final step for proofing Lemma 5.14 can be obtained by combining the results of Claim 1 and Claim 5. The function $f(n)$ was introduced as a lower bound on $P(T)$ for any full binary tree T with n leaves.

$$\Rightarrow P(T_1) = P(T_2) \geq f(n) = \Theta(n \log(n)).$$

Thus Claim 1 provides a way to construct $\mathcal{O}(n^2 \log^2(n))$ pairings of edges that violate the requirements for arboreal matchings. $\square$

6 Implementation generalized Robinson Foulds

This chapter discusses all necessary preparations and additional comments for the gRF's implementation. They include creating test instances and choosing meaningful distance function among others. Last but not least some implementation details are presented and a short summary of the results is given.
All scripts and functionalities are written in the programming language *Python*. The whole package is stored in the following Github repository:

`https://github.com/bananajoe/TreeComparison`

It includes a module to compute and store the Catalan numbers, to create and store randomized full binary trees and finally pairwise comparing them.

6.1 Preparation and Overview

The comparison of tree distances is, as already explained, a necessary tool for different research fields. In this case the gRF is only applicable to phylogenetic trees on the same set of taxa, where the inner nodes aren't labelled and don't get any attention. This behaviour has to be reflected in the cost functions for the tree edit distance. Furthermore all test instances shall be full binary trees. 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

The database of test instances had to be created because there was no suitable one available. As already discussed, it has to consist of randomly selected full binary trees with n leaves, $n \in \mathbb{N}$.
The algorithm generating this database creates full binary trees uniformly at random. To ensure uniform distribution it uses Lemma 2.27, namely that the number of full binary trees with n leaves is the $(n-1)$-th Catalan number. The algorithm recursively picks the sizes of the left and right subtrees for every inner node. In each step the sizes have to be chosen such that every feasible outcome is equally possible. The first few recursion steps shall demonstrate how the algorithm is designed:

- $\underline{n=2}$: It is obvious that there is only one full binary tree with two leaves.
- $\underline{n=3}$: The tree is automatically determined as soon as it is known whether the left subtree has one or two leaves. Therefore there are exactly two possible full binary trees with three leaves.
- $\underline{n=4}$: The root's left subtree may contain between one and three leaves. A further discussion of the implied case distinction gives an idea about how the algorithm works:
  - Case 1: The left subtree contains one leaf.
    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, ending 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 holds for the right one. 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 Case 1 $\Rightarrow$ there are two such trees.

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

The uniform distribution among all full binary trees needs to be assured. Therefore the probability of picking the size of the root's left subtree has to be chosen accordingly.
This task can be exemplified by considering the case $n = 4$. Every feasible tree has to be chosen with probability $\frac{1}{5} = \frac{1}{C_3}$. Trivially Case 2, where both subtrees contain two leaves, has to be picked with probability $\frac{1}{5}$ since this property already determines the only possible outcome. The other cases are symmetric, so both of them need to be chosen with equal likelihood of $\frac{2}{5}$. In both cases one has to choose a full binary tree with three leaves. Since there are exactly two of them, both have to be taken with probability $\frac{1}{2} = \frac{1}{C_2}$. This way every full binary tree with four leaves gets chosen with equal probability.

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

Proof. Proof by induction:
The routine CreateFullBinaryTree(1) returns the only possible full binary tree with one leaf, which gets chosen with probability 1. For the induction, suppose CreateFullBinaryTree(j) returns a full binary tree with j leaves chosen uniformly at random among all such trees where $1 \leq j < n$.
The algorithm performs the recursive step since $n > 1$. Let $p_i := \frac{C_{i-1}C_{n-1-i}}{C_{n-1}}$ be the fraction used inside the for loop.

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

Algorithm 2 Choosing a full binary tree with n leaves uniformly at random

```
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
```

Proof of Claim 1:

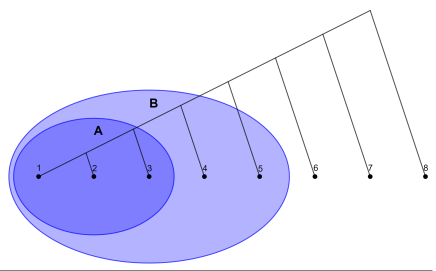
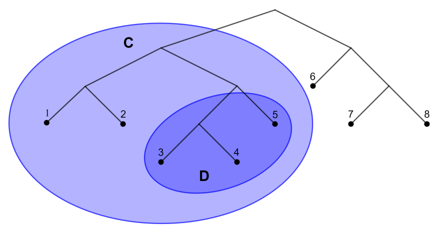
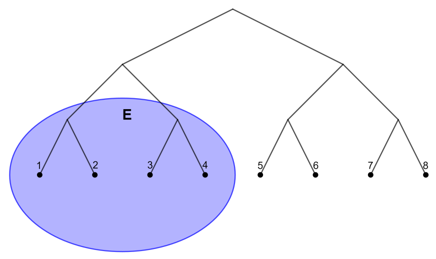
$$P = \sum_{i=1}^{n-1} p_i = \sum_{i=1}^{n-1} \frac{C_{i-1}C_{n-1-i}}{C_{n-1}} = \frac{\sum_{i=1}^{n-1} C_{i-1}C_{n-1-i}}{C_{n-1}} = \frac{\overbrace{\sum_{j=0}^{(n-1)-1} C_j C_{(n-1)-1-j}}^{C_{n-1}}}{C_{n-1}} = 1$$

Therefore the p_i's can be considered as probabilities. The algorithm's next step is choosing $r \in [0,1)$ uniformly at random, which lies within the interval $I[i]$ with probability p_i for all $1 \leq i < (n-1)$. Thus it constructs a tree, where the left subtree has i leaves, with probability $p_i = \frac{C_{i-1}C_{n-1-i}}{C_{n-1}}$. The routine recursively creates the left subtree of the root with i and a right one with $n-i$ leaves. By induction hypothesis, CreateFullBinaryTree(i) creates 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 $n-i$. All in all, the algorithm picks an arbitrary full binary tree with the following probability:

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

□

Repeatedly performing this routine built a uniformly randomized set of full binary trees with different numbers of leaves. The test instances were stored as Json-encoded lists of trees, where nodes are represented as a recursive lists with one or two elements. Take a look at the following examples:

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

6.1.2 Distance Function

The generalized Robinson-Foulds opens up the possibility to choose different distance measures. The goal is to compare these results with the tree edit distance. Therefore the best distance function is the one that counts wrong leaves, sticking with the introduced Robinson-Jaccard metric. Computing the distances between trees with different values for the constant k may show a pattern.

6.2 Implementation Details

The implementations of both the gRF and the tree edit distance is based on the data structure used in an implementation of Shasha and Zhangs algorithm [12] introduced in Section 3.3.1. Some necessary customization of this data structure *Node* include the following class properties:

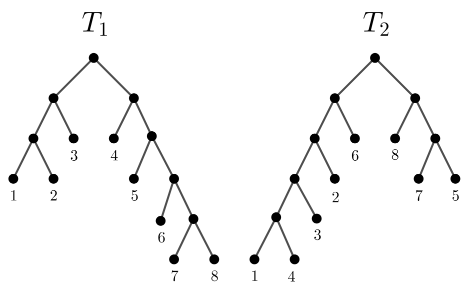
```python
from zss.simple_tree import Node

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):
        #randomly labelling the constructed trees examples
```

Listing 6.1: Scratch of the class ExtendedNode

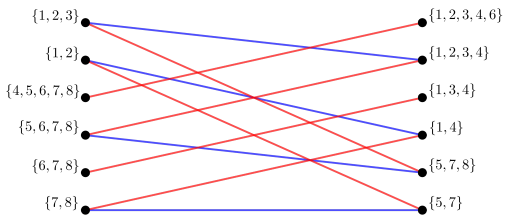
To compute the gRF, the ILP introduced in Theorem 5.13 is solved by a widely known, open-source library for linear programs within python, the PuLP-package. Details about the construction of this ILP can be found in the git repository.

6.3 Results

As already mentioned, this implementation of gRF uses the Jaccard metric to calculate the distance between sets of clusters. Varying $k \in \{1, 4, 16, 64\}$ lead to the conclusion that the gRF is directly proportional to k. Moreover there is a trend as k is increasing: The higher k gets, the less value a bigger intersection of clades has. The overall number of matched clades is more important than the quality of a matching. The difference between these types of matchings is illustrated in Figure 6.1:



(a) Two phylogenetic trees T_1 and T_2.



(b) Graph G_{T_1,T_2} shows all non trivial clusters of T_1 and T_2. The blue edges belong to the optimal quality matching, the red ones to the optimal quantity matching. The respective optimalities are independent from the value k.

Figure 6.1: A small example suggesting the importance of quantity rather than quality, as k increases.

Figure 6.1 illustrates two phylogenetic trees on the set of taxa $\{1, ..., 8\}$. Because of the restriction to arboreal matchings, there are two possible ways to map the clusters together: Mapping all clusters associated with nodes on the left subtree of T_1 to clusters corresponding either to nodes in the left or the right subtree of T_2. This leads to 2 possible matchings, M_1 and M_2 where M_1 is the optimal matching where the left subtrees are matched as well as the right ones, and M_2 being the optimal one among those where the left and right subtrees are crosswise matched. Matching M_1 has the advantage, that the quality of its matched clusters is better. Such matchings shall be called *quality* matchings. Whereas matching M_2 may have more matched nodes, but with smaller intersections among matched clusters, therefore calling it a *quantity* matching.

Interestingly, the type of an optimal matching changes as k increases. For $k = 1$, the overall optimal arboreal matching is a quality matching. Although it has 2 matched clusters less than an optimal quantity matching, it is still more efficient because of the big intersections. However as soon as $k > 1.5$, the power within the Jaccard distance decreases the value of quality to a level, that pure quantity is better than quality:

Case $k = 1$:

$$d_{JRF}^{(1)}(T_1, T_2) =$$

$$\min_{M \text{ matching}} \Bigg(\sum_{(C,C') \in M} 1 - (\frac{|C \cap C'|}{|C \cup C'|})^k + \sum_{\substack{C \in \mathcal{C}^*(T_1), \\ C \text{unmatched in } M}} 1 + \sum_{\substack{C' \in \mathcal{C}^*(T_2), \\ C' \text{unmatched in } M}} 1 \Bigg) =$$

$$\min\Bigg(\underbrace{((1-\frac{3}{4}) + (1-\frac{2}{3}) + (1-\frac{3}{4}) + (1-\frac{1}{3}) + 2 + 2)}_{\text{optimal quality matching}},$$

$$\underbrace{((1-0) + (1-0) + (1-\frac{1}{4}) + (1-0) + (1-0) + (1-0))}_{\text{optimal quantity matching}} \Bigg) =$$

$$\min\Bigg(\underbrace{5.5}_{\text{optimal quality matching}}, \underbrace{5.75}_{\text{optimal quantity matching}} \Bigg) = 5.5$$

Case $k = 2$:

$$d^{(2)}_{JRF}(T_1, T_2) =$$

$$\min\Bigg(\underbrace{((1-(\frac{3}{4})^2) + (1-(\frac{2}{3})^2) + (1-(\frac{3}{4})^2) + (1-(\frac{1}{3})^2) + 2 + 2)}_{\text{optimal quality matching}},$$

$$\underbrace{((1-0) + (1-0) + (1-(\frac{1}{4})^2) + (1-0) + (1-0) + (1-0))}_{\text{optimal quantity matching}} \Bigg) =$$

$$\min\Bigg(\underbrace{6.3194^\circ}_{\text{optimal quality matching}}, \underbrace{5.875}_{\text{optimal quantity matching}} \Bigg) = 5.875$$

The example above demonstrates the behaviour which can observed within all test instances. For any instance the gRF distance tends to maximize the number of matched cluster without taking the corresponding taxons under consideration as k increases.

Furthermore, the running time is a big problem for this approach. Lemma 5.14 provides a lower bound on the number of restrictions of at least $n^2 \log^2(n)$. This leads to a huge ILP even for small trees with only 24 leaves. Handling such instances took the university's server approximately 500 seconds. Raising the number of leaves to 32 on each tree caused a running time of over 3100 seconds per instance.
It is possible that more evolved linear programming packages solve these problems more efficiently. One of these packages is the well known LP solver Gurobi. Nevertheless, this approach doesn't seem to bring fast solutions for big problems as the number of constraints increases at a high pace.

7 Implementation Tree Edit Distance

This chapter presents Henderson's module for Shasha and Zhang's algorithm provided in his Github repository [12]. Furthermore meaningful choices for cost functions are discussed and compared with each other. The chapter is closed by an overview of the results and some general behavior found during investigating them.

7.1 Shasha and Zhang's algorithm by Henderson

Suppose two labelled trees A and B are given. Henderson's implementation of finding their TED can be split into two separated steps:

1. Finding out the post-order traversal index and the keyroots for A and B.
2. Computing the TEDs for all relevant subproblems, i.e. combinations of subtrees induced by keyroots of A and B respectively.

Ad Step 1:
Listing 7.1 presents the initialization of class AnnotatedTree. It can be split up further into two parts: Step 1a) and Step 1b), as suggested within the listing

```python
class AnnotatedTree(object):
    def __init__(self, root, get_children):
        #init 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* needed for Step 1b). Variable *stack* is of type stack, a specific data structure. This data structure is a collection of objects supporting fast last-in, first-out semantics. Throughout the process of the loop *stack* stores pairs of data: A node and a list of all its ancestors. The variable is initialized as a stack consisting of a pair of the tree's root and an empty collection, since the root doesn't have an ancestor.
While the stack still contains some pair of data its method *pop* gets executed. This function returns the last element of the stack and removes it from the stack. Every node is associated with a unique id *nid*. If the inspected node has children, *stack* gets extended with pairs of each child and the updated list of ancestors.
The essential detail is that every pair gets *appended* to the stack, thus putting it at the end of *stack*. The reason why this is so important will be explained later. After appending all its children to the stack, the investigated node and some additional data get pushed onto another stack called *pstack*. This stack will be used in Step 1b) to get the correct post-order index for every node.

Ad Step 1b):

This step determines the keyroots of the investigated tree. 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 v' of a node v has a different left most descendant than the node v itself. But any node with a higher index can not have the same left most descendant as v' because of the way all nodes are indexed. Thus the node with the highest index among all nodes sharing the same leftmost descendant, is required to be a keyroot. Two find such nodes, two temporary dictionaries *lmds*, the list of every node's left most descendant, and *keyroots*, a dictionary saving the highest index of all nodes that share a leftmost descendant, are introduced. The stack *pstack* is constructed such that popping it consecutively yields nodes in the same order as their post-order index. The counter *i* puffers the actual post-order index of a node. During every pass through the while loop the routine computes a node's left most descendant by checking different cases. Afterwards it updates a temporary dictionary *keyroots* which is used to evaluate all keyroots at the end of Step 1b).

The initialization of an AnnotatedTree makes use of efficiently appending and popping a stack to receive the correct indexing and all necessary information to get the list of keyroots.

Ad Step 2:
After Step 1), the actual computation of the TED is done by first considering all relevant subproblems. A matrix is computed storing all pairwise distances between 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 [12].

7.2 Distance measures

The simplest distance measure for the TED just counts the number of operations needed to transform one tree into the other. This implies that any insertion, deletion and renaming costs the same value of 1. This distance measure is called the *standard tree edit distance* (STED)
A big difference between the TED and the gRF is that the latter does not take the interior nodes of the trees into account. Therefore structural differences, like inserting an interior node to prolong the path at some point, don't affect the gRF immediately. However one has to perform at least an additional deletion-operation when using the TED. Every insertion and deletion of an interior node increases the TED. This leads to the first investigated adaption of operation costs: Making the insertion and deletion of interior nodes free of charge. This distance measure will later be referred to as *cheap tree edit distance* (CTED)
Another distinction is that the gRF doesn't make a difference between left and right. Therefore there was an attempt to adjust the distance measure by adapting the direction. The idea is to restructure one of the trees by swapping the order among some sibling pairings. Considering all combinations

of sibling pairings leads to $O(2^n)$ possible trees. The approach however considers only a specific restructured tree:
Let r_1 and r_2 be the roots of the inspected trees. Denote the left subtree of root r_i as L_i and the right one as R_i for $i \in \{1, 2\}$. If $|V(L_1)| > |V(R_1)|$, but the opposite holds true for the other tree, i.e. $|V(L_2)| < |V(R_2)|$, 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 gRF remains the same, but the TED may change. After having investigated the root perform the same comparison between L_1 and L_2, R_1 and R_2 respectively. Computing a TED after adapting one tree in such a way, adds the adjective *adapted* to the distance's name, i.e. *adapted standard tree edit distance* (aSTED) and *adapted cheap tree edit distance* (aCTED).
Last but not least, a compromise between the CTED and STED can be investigated by defining the costs of insertions and deletions of inner nodes as some $0 < \alpha < 1$. Assuming $\alpha = \frac{1}{2}$, this distance measure is called $\frac{1}{2}$-*alpha tree edit distance* ($\frac{1}{2}$-ATED).

7.3 Results

It was possible to compute the TED for instances with up to 256 leaves per tree. Afterwards, the problems became too big for the working memory of the university's servers. In most real life applications, two trees with up to 1000 nodes in total should suffice. Otherwise one needs to execute the calculations on more powerful computers or servers.

While comparing the different TEDs among each other, the trivial conclusion, that the STED leads to bigger distance values than the CTED or any α-ATED, was found. This is trivial as every operation is as least as expensive for the STED in comparison to the CTED or an α-ATED. Some additional notations will simplify the further discussion:

- *Test instances*, $\mathcal{T}(n)$: The generated set of all test instances where both trees have exactly n leaves. Any instance is a set of two different, randomly generated phylogenetic trees with n leaves on the set of taxa $X := \{1, ..., n\}$. Furthermore, denote $\mathcal{T}$ as the set of all test instances:

$$\mathcal{T} := \bigcup_{i \in I} \mathcal{T}(i) \quad I := \left\{ \substack{2,3,...12,16,20,24,32,40,\\ 48,64,128,192,256,512} \right\}$$

- *The average of a distance function δ, $\mathcal{A}_\delta(n)$*: Let δ be a distance function. The average $\mathcal{A}_\delta(n)$ is defined as follows:

$$\mathcal{A}_\delta(n) := \Big(\sum_{(T,T') \in \mathcal{T}(n)} d_\delta(T, T') \Big) \frac{1}{|\mathcal{T}(n)|} \frac{1}{n}$$

  This value is an indicator of the expected distance between two randomly generated phylogenetic trees with n leaves with respect to the distance function δ proportionately to n. For problem CTED this average is denoted as $\mathcal{A}_{CTED}(n)$, the other distances are denoted analogously.

The results show that the averages follow some trends:

- The average $\mathcal{A}_{CTED}(n)$ seems to converge to 1 as $n \to \infty$.
- The average $\mathcal{A}_{STED}(n)$ is strongly increasing as $n \to \infty$ and has an upper bound of 3.

The first observation can be explained by following lemmas:

Lemma 7.1. *Let T_1 and T_2 be two phylogenetic trees with n leaves on the same set of taxa. Let δ_{CTED} be the distance function used in the CTED. Then the following inequality holds true:*

$$d_{\delta_{CTED}}(T_1, T_2) \leq n$$

Proof. The distance function δ_{CTED} allows inserting and deleting any interior node without any costs. Thus one can perform a standard procedure ensuring that the overall costs are not greater than n. Figure 7.1 illustrates the aforementioned procedure for a small example.
First every inner node gets deleted until all leaves are directly connected

to the root. As any deletion is free of charge, this step costs 0 overall. Afterwards inner nodes get inserted until the structure of the adapted tree is the same as the one of tree T_2. This step is free of charge as well. The only thing left to do is relabelling all leaves to match tree T_2. The costs for this relabelling step is equal to the number of nodes that have to be relabelled. Thus the upper bound of n holds trivially. □

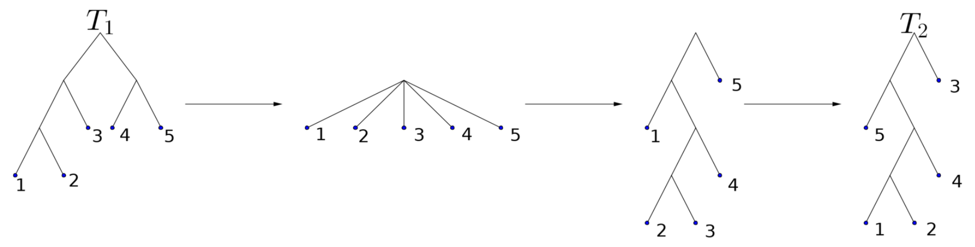


Figure 7.1: A step guide for limiting the CTED by the number of leaves.

Remark. To calculate the actual average $\mathcal{A}_{CTED}(n)$ and whether it converges to 1, one has to solve the following equation:

$$\mathcal{A}_{CTED}(n) = (\sum_{i=0}^{n} n - i\,\mathbb{P}(\pi \text{ has exactly } i \text{ fixpoints}))\frac{1}{n}$$

where π is an arbitrarily chosen permutation on the set $\{1, ..., n\}$.

Remark. The upper bound in the second observation can be proven by the same procedure described in Lemma 7.1. The only difference is, that the first two steps both cost exactly $n-2$, as this is the number of interior nodes other than the root.

These two observations resemble the data found during the computations. There are many cases where the procedure from Lemma 7.1 does not yield the best solution. For trees T_1 and T_2 in Figure 5.3 for example, the cheapest way to manipulate T_1 is to delete the leave with label 1 and its parent, insert an interior node on the edge of the root to leave 8 and append leaf 1 as another child. Thus the overall costs of this manipulation are 2.

However, in most cases the observed costs are very close to the costs of the above mentioned procedure. All these considerations are based on the fact, that the actual costs of the CTED only depend on the permutation of the leaves' labels, not on the actual structure of the trees. Two completely differently structured trees may have a CTED distance of 0 as long as the underlying permutation of the leaves' labels is the same one with respect to the left to right order. Thus the CTED can be seen as a distance measure on permutations, but it does not reflect useful information within the context of structured trees.
For the STED there are often better solutions than the plain procedure of deleting and inserting all interior nodes. Thus it computes more advanced solutions that indeed take the structure of both trees into account. On the other hand, this distance measure does not give a benefit to interior nodes. Thus it is not useful for simulating the gRF, since it doesn't concentrate on the clades and their leaves' labels.

Last but not least, varying the value of α for the α-ATED indicates that the average $\mathcal{A}_{\alpha-ATED}(n)$ is directly proportional to α. There is a trivial upper bound on this average given by:

$$\mathcal{A}_{\alpha-ATED}(n) < 1 + 2\alpha.$$

This bound can be obtained by the same procedure as described in Lemma 7.1. Furthermore it complies with following properties:

$$\begin{aligned} \mathcal{A}_{\alpha-ATED}(n) &\longrightarrow \mathcal{A}_{CTED}(n) && \text{as } \alpha \to 0 \\ \mathcal{A}_{\alpha-ATED}(n) &\longrightarrow \mathcal{A}_{STED}(n) && \text{as } \alpha \to 1 \end{aligned}$$

8 Comparing the Tree Edit Distance and the generalized Robinson Foulds Distance

This chapter compares the results of both computations. The focus of this thesis is to find a possibility to resemble the gRF, which is quite time consuming, with a special case of the TED. The chapter starts by discussing the time complexity of both measures. Afterwards the differences between the distances themselves are explained. Finally the results are presented, showing that there is no significant correlation between these two measures.

8.1 Time Complexity

The time complexity of both distance measure were mentioned briefly in the previous two chapters. The expected behaviour is reflected by the observations. Computing the TED was possible for trees with up to about 256 leaves each in a manageable time of approximately 520 seconds. Although it was possible to create bigger instances with up to 512 leaves each, the computation of a distance never ended. The time to compute the TED is pretty similar for any of the investigated distance measures. This was roughly the same time as it took to find the gRF of trees with 24 leaves each.

The graph shows, that as soon as the instances reach a size of about 20 leaves each, the time to compute the gRF distance explodes. Depending on the computational possibilities, it might not be possible to compute real life

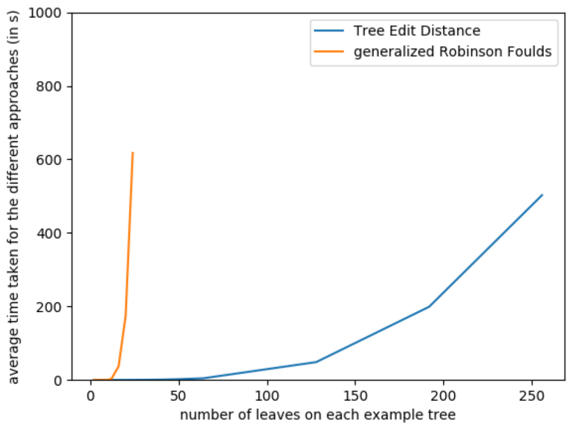


Figure 8.1: This graph shows the time it took to compute the distance measures between two trees with up to 256 leaves.

instance where the trees have for example 50 leaves. On the other hand the time to compute larger instances of the TED remains manageable.

8.2 Discussing the Results

In the previous chapter it was discussed that the CTED is mainly concentrating on the permutation of the leaves' labels, not on the tree's structure. On the other hand, the STED is not representative, as the costs are very much dependent on changes among the interior nodes, whereas they don't impact the gRF directly. Thus the $\frac{1}{2}$-ATED, which compromises the advantages of the others, is the most comparable TED to the gRF.
Since the gRF is very time consuming, the comparison of the respective distances uses instances having 20 leaves each. Because of the manageable time, it was possible to generate 300 test instances and compute their $\frac{1}{2}$-ATED and gRF distance

The $\frac{1}{2}$-ATED is much easier to compute than the gRF for a given instance. Being able to "simulate" the gRF by computing the $\frac{1}{2}$-ATED instead could save a lot of time and resources. The goal is to find any similarities or dissimilarities to show whether such a simulation would be representative. The following questions summarize possible connections between the two measures:

1. Does a low gRF imply a low $\frac{1}{2}$-ATED?
2. Does a low gRF require a low $\frac{1}{2}$-ATED?
3. Does a high gRF imply a high $\frac{1}{2}$-ATED?
4. Does a high gRF require a high $\frac{1}{2}$-ATED?

The following examples suggest potential dissimilarities:

Example low gRF, high $\frac{1}{2}$-ATED:
The first example is illustrated in Figure 8.2. It is obvious, that T_1 and T_2 have the same clusters, therefore the gRF has to be 0. But since the order of the leaves' labels is reversed, the shortest $\frac{1}{2}$-ATED is 8 and can be obtained by relabelling all leaves.

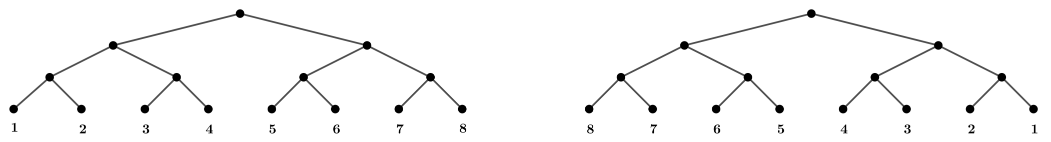


Figure 8.2: Two trees T_1 and T_2 which have a low gRF but a high $\frac{1}{2}$-ATED.

Example low $\frac{1}{2}$-ATED, high gRF:
The second example is illustrated in Figure 8.3. The value of the $\frac{1}{2}$-ATED amounts to 4.5 which can be obtained by 5 deletions and 4 insertions. On the other hand, the gRF adds up to 8. This is the highest gRF between any two trees with just 8 leaves found. Additionally, there is no randomly generated pair of trees that has a similar gRF distance in the test database. That leads to the conclusion that 8 is a among the highest gRFs for trees with 8 leaves. On the same time, the $\frac{1}{2}$-ATED is a comparatively small distance.

These two examples shall give an insight into the problems with comparing the $\frac{1}{2}$-ATED with the gRF. They show that it is possible to construct examples, that negate any similarity between the gRF and the $\frac{1}{2}$-ATED.

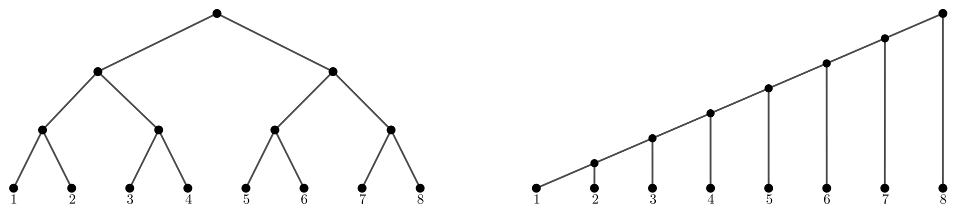


Figure 8.3: Two trees T_1 and T_2 which have a low gRF but a high $\frac{1}{2}$-ATED.

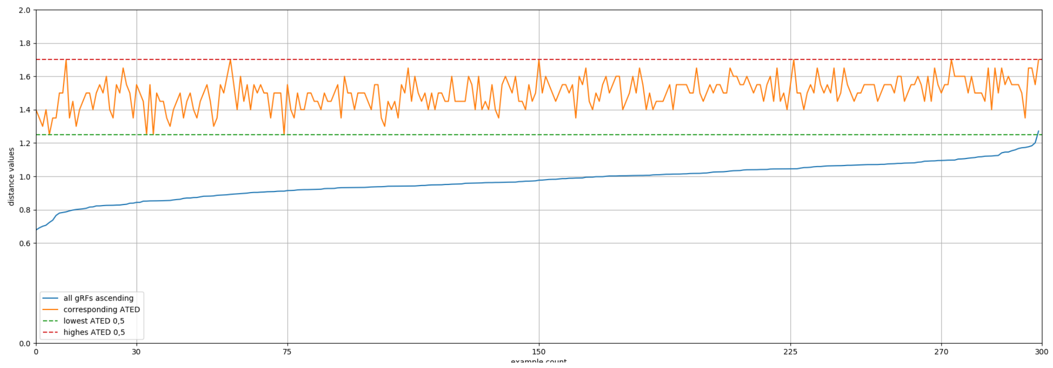


Figure 8.4: Illustration of the distances of our randomly generated test instances, sorted increasingly with respect to the gRF distance.

The test database consists of 300 pairs of randomly generated phylogenetic trees with 20 leaves. In Figure 8.4 every integer value on the X-axis corresponds to an example pairing. The instances are ordered according to their gRF increasingly. The instance at $x = 1$ has the lowest the one at $x = 300$ the highest gRF. There are additional grid lines at the 10%-, 25%-, 50%-, 75%- and 90%-marks of the gRF. Dashed lines marks the highest and lowest values of all $\frac{1}{2}$-ATED. They help identifying low and high $\frac{1}{2}$-ATED

values.
A short look on the lowest and highest 10% of all gRF values provides answers to the questions asked before. Figure 8.5 shows a magnification of these outermost parts of the graph. These results suggest, that a short gRF distance does not correlate to a short or large $\frac{1}{2}$-ATED. Actually, the randomized test instances with short gRF distances realized one of the lowest as well as one of the highest $\frac{1}{2}$-ATED, negating questions one and two. A low gRF neither implies nor requires a low $\frac{1}{2}$-ATED.
The other end of the gRF spectrum shows more correlation. No pair of trees, having a higher gRF distance than 50% of all cases, has an $\frac{1}{2}$-ATED close to the lowest distances among all test instances. However, the range of $\frac{1}{2}$-ATEDs still spans a large interval. Thus the remaining questions whether a high gRF implies or requires a high $\frac{1}{2}$-ATED has to be negated as well.

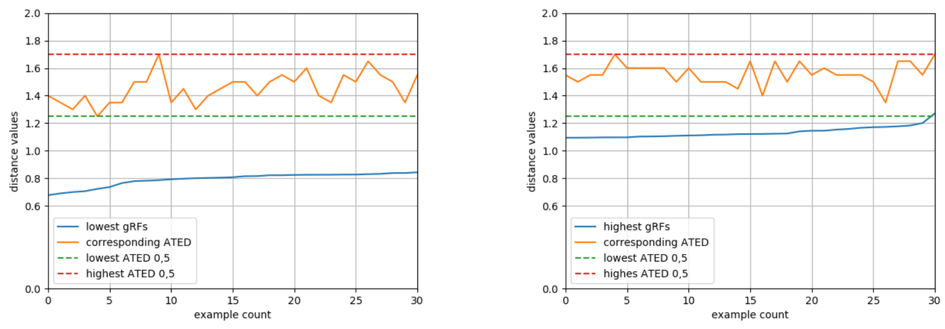


Figure 8.5: Magnifying on the extremal cases of all test instances, namely the lowest and highest 10% of gRF distances, suggests that there exists no correspondence between the gRF and the TED.

In Figure 8.6 the examples are sorted according to their $\frac{1}{2}$-ATED. This graph further shows, that there is no direct correlation between two trees' $\frac{1}{2}$-ATED and their gRF distance. The overall tendency is similar meaning that a higher $\frac{1}{2}$-ATED suggests a higher gRF but there exists no significant correspondence.

All in all, a higher gRF suggests a higher $\frac{1}{2}$-ATED and the other way around. Nevertheless, there is no significant correlation between these two

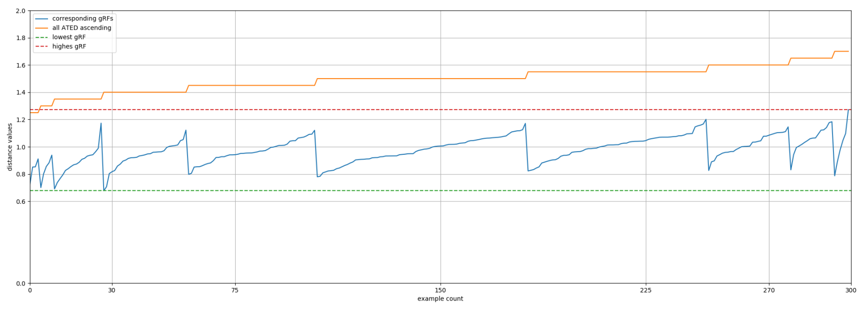


Figure 8.6: Illustration of the distances of our randomly generated test instances, sorted increasingly with respect to the $\frac{1}{2}$-ATED distance.

distance measures. Therefore it can't be possible to simulate the highly complex gRF distance with the easier $\frac{1}{2}$-ATED as a representative of all TEDs.

9 Conclusion

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

It starts off by providing basic notations and definitions since the language varies greatly between different papers and authors.

The first idea of how to compare 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 total 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. The first presented algorithm is by Shasha and Zhang [17], who use a trivial decomposition strategy guiding an iterative, recursive routine that always compares the rightmost subtrees. Afterwards two algorithms of Klein [13] and Demaine et al. [6] are presented. These algorithms involve more sophisticated decomposition strategies which take the sizes of the investigated subtrees into account. The latter even satisfies the lower bound on the running time for computing the tree edit distance with a dynamic programming approach.

The tree edit distance can be extended to the so called flexible tree matching. The idea is to relax the restrictions on ancestry and sibling groups. Penalizing such violations instead of forbidding them opens up a lot of possibilities. Computing this measure is equivalent to finding a minimum cost matching in a corresponding graph. Therefore computing the flexible tree edit distance is a strongly $\mathcal{NP}$-complete problem. The chapter includes a model for approximation heuristics. Kumar et al. [14] provide a Monte Carlo algorithm to compute an approximation of the flexible tree matching.

The third distance measured discussed is the Robinson Foulds metric, which is widely used in phylogenetics. This particular distance measure can only be applied to compare evolutionary trees on the same set of taxa. It determines the number of clades present in exactly one of the two investigated trees. It has well known advantages and disadvantages and can be generalized to use more evolved cost functions. However, computing this distance is equivalent to the $\mathcal{NP}$-complete problem of finding a minimum cost arboreal matching between the sets of non-trivial clades.

Since finding the generalized Robinson Foulds distance is $\mathcal{NP}$-complete, computing it takes a lot of time, even for small instances. Therefore it would be great to be able to simulate it with a tree edit distance algorithm. This idea is tested in a series of computational experiments.
Moreover all implementation details are presented. They contain a prove of randomness of the test instances, the creation of a binary linear program to compute the generalized Robinson Foulds metric and the implementation of Shasha and Zhang's algorithm.

The thesis ends with an analysis of the computational results. It demonstrates how much faster the tree edit distance can be computed and discusses the possible similarities of these two distance measures. The computational experiments lead to the conclusion, that the two compared distance measures are very much independent from one another. There is a small correlation, that a higher generalized Robinson Foulds distance also suggests a higher tree edit distance. Nevertheless the results show that there can't be a direct proportional correlation between them.

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] Figueiró H. V., Gang L., et al., *Genome-wide signatures of complex introgression and adaptive evolution in the big cats*, Science Advances Vo 3 no. 7, 2017

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

[10] Gusfield D., *Algorithms on strings, trees and sequences: computer science and computational biology*, Cambridge University Press, 1997

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

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

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

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

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

[16] Robinson D.F., Foulds L.R., *Comparison of phylogenetic trees*, Mathematical Biosciences Volume 53 Issues 1–2, 1981, 131–147

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

[18] Steger A., *Diskrete Strukturen: Band 1: Kombinatorik, Graphentheorie, Algebra*, Springer-Verlag, 2007

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

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

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

[22] Waterman R. A., *Introduction to computational biology: maps, sequences and genomes*, Chapman and Hall, 1995