Comenius University in Bratislava
Faculty of Mathematics,
Physics and Informatics
Department of Computer Science

Edit distance on trees and their representations

(diploma thesis)

David Zachar

Bratislava, 2010

Comenius University in Bratislava
Faculty of Mathematics,
Physics and Informatics
Department of Computer Science

Edit distance on trees and their representations

(diploma thesis)

David Zachar

Supervisor: prof. RNDr. Branislav Rovan, PhD. Bratislava, 2010

I hereby declare that I wrote this thesis by myself, with the help of the referenced literature, under the careful supervision of my thesis adviser.

. . . .

David Zachar

Abstract

The goal of this thesis is to study information about the difference between two trees which we can obtain from the difference between their string representations. First we focus on ideas of computing this difference between two trees called distance presented in other papers. Then we discuss particular string encodings of trees. In this thesis we define a new string encoding of trees. Relation between tree distance of two trees and string distance of their representation given by this encoding is shown by proofs of lower and upper bounds for this encoding. From particular encoding of trees to strings it looks like we cannot get the exact information about tree distance from string distance of its representations. This thesis contains proof that for every encoding ψ, satisfying some natural properties, it is impossible to compute the exact tree distance from the string distance of codes coded by ψ. Formally, there exist trees F and G such that $\tau(F, G) = \Omega(h)\delta(\psi(F), \psi(G))$ is true, where τ is a tree distance, δ is a string distance and h is the minimal height of trees F and G.

Acknowledgements

I would like to thank my thesis adviser Branislav Rovan for careful guidance, very useful advices and helpful discussions during the work on this thesis.

Contents

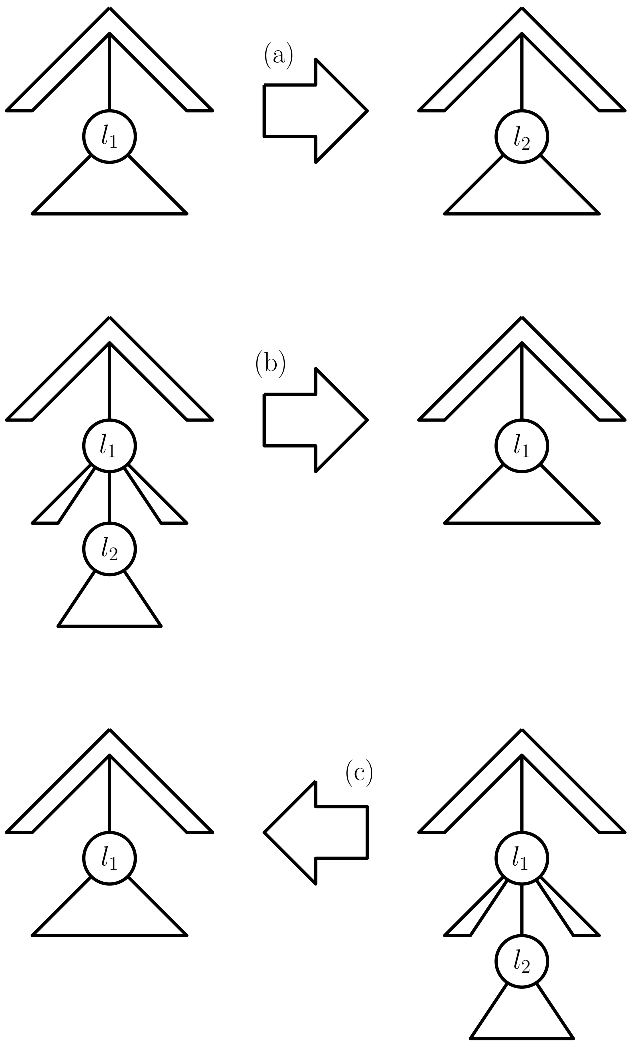
Chapter 1

Introduction

The problem of comparing trees occurs in several diverse areas such as comparing XML documents, tree data structures, molecules in chemistry, closed shapes by representing them as a tree and others. Trees are very often stored as strings and we mentioned the example of XML documents. This thesis is focusing on what information we can get about the difference between two trees from studying differences between their string representations.

The measure used for computing the difference between two trees is often the tree edit distance. The tree edit distance is extended from the string edit distance. The string edit distance is a well studied problem. The string edit distance computes the distance or difference between two strings. The tree edit distance was introduced by Tai in the late 1970's [1]. Let F and G be two rooted treed with a left-to-right order among siblings and where each vertex is assigned a label from an alphabet Σ_T. The tree edit distance between F and G is the minimum cost of transforming F into G by a sequence of elementary operations consisting of deleting and relabeling existing nodes as, well as inserting new nodes (allowing at most one operation at a time to be performed on a node). These operations are illustrated by Figure 1.1. Formally, given a node $v \in F$ with parent v', relabel changes the label of v, delete removes a non-root node v and sets the children of v as the children of v' (the children are inserted in the place of v as a sub-sequence in the left-

Figure 1.1: Elementary edit operations on vertex of tree. (a) Relabeling of the node (b) Deleting the node (c) Inserting a node



to-right order of the children of v'), and insert (the complement of delete) connects a new node v as a child of some v' in F making v the parent of a consecutive sub-sequence of the children of v'. The well researched difference of trees uses cost of elementary operations equal to 1. In this thesis we also use this definition. So the distance between two trees is the number of elementary operations, necessary to transform one tree into the other.

The problem studied in this thesis is the relation between the tree edit distance and the string edit distance of strings representing the trees. The string edit distance is defined similarly to the tree edit distance. Let us consider strings over finite or infinite alphabet Σ_S. For string s and integer $i, s[i]$ denotes the i-th character of the string s, $s[i...j]$ denotes $s[i]...s[j]$, and $|s|$ denotes the length of s. The edit distance between two strings s_1 and s_2 is defined as the minimum number of operations to transform s_1 into s_2. In this paper $\delta(s_1, s_2)$ is used to denote the edit distance between s_1 and s_2.

This thesis starts by a survey of results from other papers. Chapter 2 contains short description of ideas for computing tree edit distance. How these ideas helped to improve algorithms for computing tree edit distance. In Chapter 3 we start to focus on encoding trees to strings. The first part of this chapter is dedicated to encodings from other works, but then we introduce new encoding with upper and lower bound proofs for it. At the end of this chapter we present the worst case scenario for these encodings in relation to information about the tree edit distance. Chapter 4 contains proof of a generalized property of codings that we could see in particular codings. Unfortunately this natural property of encodings allows us to prove that for nice encodings the string distance of the codes will not give us exact information about the tree distance of the trees represented.

Chapter 2

Tree edit distance

To compute the tree edit distance faster there has been made a lot of research. As we can see on the Table 2.1 the first algorithm described in 1979 by Tai in [1] needed $O(n^6)$ time in the worst case. However recent results in [5] show that computing the tree edit distance can by done in $\Theta(n^3)$.

This work is focusing on relation between tree edit distance and string edit distance of string representations of that trees. The result from [5] that tree edit distance can be computed in $\Omega(n^3)$ tells us something about the relation. Because string edit distance could be computed in $O(n^2)$ and even better. At first it tells us the very obvious proposition that tree structure stores something more valuable then string structure. Also these results show us that we should not be able to construct ideal string representation of a tree. In this context ideal string representation of two trees will have the same string distance as the tree distance of the coded trees for every two trees. If such a representation of a tree exists it should be very hard to construct or the process of coding trees to this ideal representation and computing tree edit distance is somehow omitting the Decomposition strategy used in the proof of bound $\Omega(n_3)$ for time needed to compute tree edit distance. Hard to construct means $\Omega(n^3)$ of time complexity to compute string representation of a tree. These are the only two ways for that the lower bound could be true.

Table 2.1: Results of Time and Space cost for computing Tree edit distance

Author [Reference]	**Time**	**Space**	**Worst Time**
Tai [1]	$O(n_l^2 m_l^2 nm)$	$O(n_l^2 m_l^2 nm)$	$O(n^6)$
Shasha and Zhang [2]	$O(n_{hl} m_{hl} nm)$	$O(nm)$	$O(n^4)$
Klein [3]	$O(m^2 n \log n)$	$O(nm)$	$O(n^3 \log n)$
Duluq and Touzet [4]	$\Omega(nm \log n \log m)$		$\Omega(n^2 \log^2 n)$
Demaine at al. [5]	$O(nm^2(1 + \log \frac{n}{m})$	$O(nm)$	$O(n^3)$
Demaine at al. [5]	$\Omega(nm^2(1 + \log \frac{n}{m})$		$\Omega(n^3)$

Notation:
n, m - sizes of input trees F, G ordered so that $n \geq m$
n_l, m_l - number of leaves in F, G; n_h, m_h - height of F,G
$n_{hl} = min\{n_h, n_l\}$, $m_{hl} = min\{m_h, m_l\}$

Because of such a strong relation between these results and the main goal of this thesis, this chapter contains the main ideas of these results. Here is only a survey of ideas and algorithms. Better detailed algorithms and proofs can be found in the previous works that are listed in Bibliography.

2.1 Shasha and Zhang's Algorithm [2]

Given two forests F and G of sizes n and m respectively, the following lemma is easy to verify. Intuitively, the lemma says that in any sequence of edit operations the two rightmost roots in F and G must either be matched with each other or else one of them is deleted.

Lemma 1. $\tau(F, G)$ *can be computed as follows:*
$\tau(\emptyset, \emptyset) = 0$
$\tau(F, \emptyset) = \tau(F - r_F, \emptyset) + c_{del}(r_F)$
$\tau(\emptyset, G) = \tau(\emptyset, G - r_G) + c_{del}(r_G)$

$$\tau(F,G) = \min \begin{cases} \tau(F - r_F, G) + c_{del}(r_F), \\ \tau(F, G - r_G) + c_{del}(r_G), \\ \tau(R_F^\circ, R_G^\circ) + \tau(F - R_F, G - R_G) + c_{match}(r_F, r_G) \end{cases}$$

Notation: The unique empty forest/tree is noted $\emptyset$. F_v denotes the sub-tree of F rooted at v. $F - v$ denotes the forest F after deleting v. The special case of $F - root(F)$ where F is a tree and $root(F)$ is its root is denoted F°. The leftmost and rightmost trees of a forest F are denoted L_F and R_F and their roots l_F and r_F. $F - L_F$ denotes the forest F after deleting the entire left most tree L_F; similarly $F - R_F$.

Lemma 1 yields an $O(m^2n^2)$ dynamic program algorithm. If we index the vertices of the forests F and G according to their left-to-right postorder traversal position, then entries in the dynamic program table correspond to pairs (F', G') of subforests F' of F and G' of G where F' contains vertices $\{i_1, i_1 + 1, \dots, j_1\}$ and G' contains vertices $\{i_2, i_2 + 1, \dots, j_2\}$ for some $1 \leq i_1 \leq j_1 \leq n$ and $1 \leq i_2 \leq j_2 \leq m$.

However, as we will presently see, only $O(\min\{n_h, n_l\} \min\{m_h, m_l\} nm)$ different relevant subproblems are encountered by the recursion computing $\tau(F, G)$. We calculate the number of relevant subforests of F and G independently, where a forest F' (respectively G') is a relevant subforest of F (respectively G) if it occurs in the computation of $\tau(F, G)$. Clearly, multiplying the number of relevant subforests of F and of G is an upper bound on the total number of relevant subproblems.

We now count the number of relevant subforests of F; the count for G is similar. First, notice that for every node $v \in F$, F_{v_O} is a relevant subproblem. This is because the recursion allows us to delete the rightmost root of F repeatedly until v becomes the rightmost root; we then match v (i.e., relabel it) and get the desired relevant subforest. A more general claim is stated and proved later on in Lemma 2. We define

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

It is easy to see that every relevant subforest of F is a prefix (with respect to the postorder indices) of F_v° for some node $v \in keyroots(F)$. If we define *cdepth*(v) to be the number of keyroot ancestors of v, and *cdepth*(F) to be the maximum *cdepth*(v) over all nodes $v \in F$, we get that the total number of relevant subforest of F is at most

$$\sum_{v \in keyroots\{F\}} |F_v| \leq \sum_{v \in F} cdepth(v) \leq \sum_{v \in F} cdepth(F) = |F| cdepth(F)$$

This means that given two trees, F and G, of sizes n and m we can compute $\tau(F, G)$ in $O(cdepth(F)cdepth(G)nm) = O(n_h m_h nm)$ time. Shasha and Zhang also proved that for any tree T of size n, $cdepth(T) \leq \min\{n_h, n_l\}$, hence the result. In the worst case, this algorithm runs in $O(m^2n^2) = O(n^4)$ time.

2.2 Klein's algorithm [3]

Klein's algorithm is based on a recursion similar to Lemma 1. Again, we consider forests F and G of sizes $|F| = n \geq |G| = m$. Now, however, instead of recursing always on the rightmost roots of F and G, we recurse on the leftmost roots if $|L_F| \leq |R_F|$ and on the rightmost roots otherwise. In other words, the "direction" of the recursion is determined by the (initially) larger of the two forests. We assume the number of relevant subforests of G is $O(m^2)$; we have already established that this is an upper bound.

We next show that Klein's algorithm yields only $O(n \log n)$ relevant subforests of F. The analysis is based on a technique called heavy path decomposition introduced by Harel and Tarjan [6]. Briefly: we mark the root of F as light. For each internal node $v \in F$, we pick one of v's children with maximal number of descendants and mark it as heavy, and we mark all the other children of v as light. We define *ldepth*(v) to be the number of light nodes that are ancestors of $v \in F$, and *light*(F) as the set of all light nodes in F . By [6], for any forest F and vertex $v \in F$, $ldepth(v) \leq \log |F| + O(1)$. Note that every relevant subforest of F is obtained by some $i \leq |F_v|$ consecutive deletions from F_v for some light node v. Therefore, the total number of relevant subforests of F is at most

$$\sum_{v \in light(F)} |F_v| \leq \sum_{v \in F} 1 + ldepth(v) \leq \sum_{v \in F} (\log(|F|) + O(1)) = O(|F| \log |F|)$$

Thus, we get an $O(m^2 n \log n) = O(n^3 \log n)$ algorithm for computing $\tau(F, G)$.

2.3 Decomposition Strategy Framework [4]

Both Klein's and Shasha and Zhang's algorithms are based on Lemma 1. The difference between them lies in the choice of when to recurse on the rightmost roots and when on the leftmost roots. The family of decomposition strategy algorithms based on this lemma was formalized by Dulucq and Touzet in [4].

Definition 1. Let F and G be two forests. A strategy is a mapping from pairs (F, G) of subforests of F and G to set $\{\textit{left}, \textit{right}\}$. A decomposition algorithm is an algorithm based on Lemma 1 with the directions chosen according to a specific strategy.

Each strategy is associated with a specific set of recursive calls (or a dynamic programming algorithm). The strategy of Shasha and Zhang's algorithm is $S(F', G') = right$ for all F' , G' . The strategy of Klein's algorithm is $S(F', G') = left$ if $|L_{F'}| \leq |R_{F'}|$, and $S(F', G') = right$ otherwise. Notice that Shasha and Zhang's strategy does not depend on the input trees, while Klein's strategy depends only on the larger input tree. Dulucq and Touzet proved a lower bound of $\Omega(mn \log m \log n)$ time for any decomposition strategy algorithm.

The following lemma states that every decomposition algorithm computes the edit distance between every two root-deleted subtrees of F and G.

Lemma 2. *Given a decomposition algorithm with strategy S, the pair (F_v°, G_w°) is a relevant subproblem for all $v \in F$ and $w \in G$ regardless of the strategy S.*

Proof. First note that a node $v' \in F_v$ (respectively, $w' \in G_w$) is never deleted or matched before v (respectively, w) is deleted or matched. Consider the following specific sequence of recursive calls:

- Delete from F until v is either the leftmost or the rightmost root.
- Next, delete from G until w is either the leftmost or the rightmost root.

Let (F', G') denote the resulting subproblem. There are four cases to consider.

(1) v and w are the rightmost (leftmost) roots of F' and G', and $S(F', G') = right(left)$. Match v and w to get the desired subproblem.

(2) v and w are the rightmost (leftmost) roots of F' and G', and $S(F', G') = left(right)$. Note that at least one of F', G' is not a tree (since otherwise this is case (1)). Delete from one which is not a tree. After a finite number of such deletions we have reduced to case (1), either because S changes direction, or because both forests become trees whose roots are v, w.

(3) v is the rightmost root of F', w is the leftmost root of G'. If $S(F', G') = left$, delete from F' ; otherwise delete from G'. After a finite number of such deletions this reduces to one of the previous cases when one of the forests becomes a tree.

(4) v is the leftmost root of F', w is the rightmost root of G'. This case is symmetric to (3).

□

2.4 Demaine et al.'s Algorithm [5]

Demaine et al.'s algorithm for computing $\tau(F, G)$ given two trees F and G of sizes $|F| = n \geq |G| = m$ recursively uses a decomposition strategy in a divide-and-conquer manner to achieve $O(nm^2(1 + \log m)) = O(n^3)$ running time in the worst case. For clarity I describe the algorithm recursively and analyze its time complexity. Demain et al. proved that the space complexity of a bottom-up non recursive implementation of the algorithm is $O(mn) = O(n^2)$.

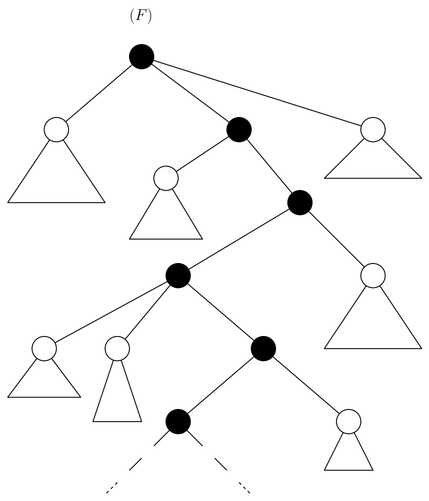
Before presenting this algorithm, let me try to develop some intuition. We begin with the observation that Klein's strategy always determines the direction of the recursion according to the F-subforest, even in subproblems where the F-subforest is smaller than the G-subforest.

However, it is not straightforward to change this since even if at some stage we decide to choose the direction according to the other forest, we must still make sure that all subproblems previously encountered are entirely solved. At first glance this seems like a real obstacle since apparently we only add new subproblems to those that are already computed. Our key observation is that there are certain subproblems for which it is worthwhile to choose the direction according to the currently larger forest, while for other subproblems we had better keep choosing the direction according to the originally larger forest.

The heavy path of a tree F is the unique path starting from the root (which is light) along heavy nodes. Consider two trees, F and G, and assume we are given the distances $\tau(F_v^\circ, G_w^\circ)$ for all $v \in F$ and $w \in G$. By Lemma 2, these are relevant subproblems for any decomposition strategy algorithm. How would we go about computing $\tau(F, G)$ in this case? Using Shasha and Zhang's strategy would require $O(|F||G|)$ time, while using Klein's strategy would take $O(|F||G|^2)$ time. Let us focus on Klein's strategy since Shasha and Zhang's strategy is independent of the trees. Note that even if we were not given the distance $\tau(F_v^\circ, G_w^\circ)$ for a node u on the heavy path of F , we would still be able to solve the problem in $O(|F||G|^2)$ time. To see why, note that in order to compute the relevant subproblem $\tau(F_u, G_w)$, we must compute all the subproblems required for solving $\tau(F_v^\circ, G_w^\circ)$ even if $\tau(F_v^\circ, G_w^\circ)$ is given.

Demaine et al. defines the set $TopLight(F)$ to be the set of roots of the

Figure 2.1: A tree F with n nodes.The black nodes belong to the heavy path. The white nodes are in $TopLight(F)$, and the size of each subtree rooted at a white node is at most $\frac{n}{2}$. Note that the root of the tree belongs to the heavy path even thought it is light.



forest obtained by removing the heavy path of F . Note that $TopLight(F)$ is the set of light nodes with $ldepth(1)$ in F (see the definition of $ldepth$ in Klein's algorithm section). This definition is illustrated in Figure 2.1. It follows from Lemma 2 that if we compute $\tau(F_v, G)$ for all $v \in TopLight(F)$, we would also compute all the subproblems $\tau(F_v^\circ, G_w^\circ)$ for any $w \in G$ and v not on the heavy path of F. Note that Klein's strategy solves $\tau(F_v, G)$ by determining the direction according to F_v even if $|F_v| < |G|$. We observe that we can do better if in such cases we determine the direction according to G. It is important to understand that making the decisions according to the larger forest when solving $\tau(F_v^\circ, G_w^\circ)$ for any $v \in F$ and $w \in G$ (i.e., regardless of whether v is on the heavy path or not) would actually increase the running time. The identification of the set $TopLight(F)$ is crucial for obtaining the improvement.

Given these definitions, the recursive formulation of Demain et al.'s algorithm is simply:

Algorithm $\tau(F, G)$ can be computed recursively as follows:

(1) If $|F| < |G|$, compute $\tau(G, F)$ instead.

(2) Recursively compute $\tau(F_v, G)$ for all $v \in TopLight(F)$.

(3) Compute $\tau(F, G)$ using the following decomposition strategy: $S(F, G) = left$ if F is a tree, or if F is not the heavy child of its parent. Otherwise, $S(F, G) = right$. However, do not recurse into subproblems that were previously computed in step (2).

The algorithm's first step makes sure that F is the larger forest, and the second step makes sure that $\tau(F_v^\circ, G_w^\circ)$ is computed and stored for all v not in the heavy path of F and for all $w \in G$. Note that the strategy in the third step is equivalent to Klein's strategy for binary trees. For higher valence trees, this variant first makes all left deletions and then all right deletions, while Klein's strategy might change direction many times. They are equivalent in the important sense that both delete the heavy child last. The algorithm is evidently a decomposition strategy algorithm, since for all subproblems, it either deletes or matches the leftmost or rightmost roots. The correctness of the algorithm follows from the correctness of the decomposition strategy algorithms in general.

Demaine et al. proved that this algorithm is running in $O(nm^2(1+\log\frac{n}{m}))$ time and $O(nm)$ space. They also proved the tighter lower bound $\Omega(nm^2(1+\log\frac{n}{m}))$ for any decomposition based algorithm. So the algorithm is the best time running algorithm in decomposition based algorithms for computing tree edit distance. The proofs can be found in [5].

Chapter 3

Encoding trees by strings

Almost every tree structure we store as a string. In computer memory or in hard disk, the tree structures are stored as a sequence of memory blocks or characters. The best example is that of XML documents which are widely used. So it is obvious we have thought let us compute distance of this representations. However from the previous chapter we know this is not what will be working exactly. It looks we can not compute the exact tree distance from distance of their representations.

There has been made a research in this area. We can tell that string distance of coded trees is something different then tree distance of these trees for all the researched codings. However these two distances are tied together but no too tight. The relation between these distances on some researched codings is shown in Table 3.1.

This chapter contains short description of these encodings. First three encodings namely Euler String, modified Euler String and Binary Tree Code 1 are researched in [7, 10, 12]. In these references the proofs of bounds presented in Table 3.1 can be found. The last encoding Binary Tree Code 2 is new and is defined in this thesis. So this chapter contains description of this coding and proofs of bounds for this coding too. End of this chapter is given to description of tightness of upper bounds for all codings mentioned here except the modified Euler String. The worst case of trees for upper

Table 3.1: Bounds of tree representations

Name of coding	**Lower and Upper Bounds**
Euler string	$\frac{1}{2}\delta(\psi(F),\psi(G)) \leq \tau(F,G) \leq (2h+1)\delta(\psi(F),\psi(G))$
mod. Euler string	$\frac{1}{6}\delta(\psi(F),\psi(G)) \leq \tau(F,G) \leq O(n^{\frac{3}{4}})\delta(\psi(F),\psi(G)$
Binary tree code 1	$\frac{1}{2}\delta(\psi(F),\psi(G)) \leq \tau(F,G) \leq (h+1)\delta(\psi(F),\psi(G)) + h$
Binary tree code 2	$\frac{1}{2}\delta(\psi(F),\psi(G)) \leq \tau(F,G) \leq 2(h+1)\delta(\psi(F),\psi(G)) + 2h$

Notation:
$\psi(T)$ is the corresponding coding function of tree T to string. For example in lower and upper bounds for the Euler String ψ means ψ_E

bound of these codes is the same so it is described together for all codings.

Before the description of particular codings, we should say what coding and code is. It is good to properly define what we are meaning by these words.

Definition 2 (Coding, Coding function)**.** Let $\mathcal{T}$ be set of rooted, ordered trees with labels on vertices from alphabet Σ_T. Let $\mathcal{S}$ be a set of strings with letters from alphabet Σ_S. Function ψ defined as:

$$\psi : \mathcal{T} \to \mathcal{S}$$

we will call coding function of a tree (shortly coding function, or simply coding) if:

1. ψ is an injective function
2. ψ is computable
3. ψ^{-1} is computable if the argument is a string $\psi(T)$ for some tree T

In what follows ψ is also called encoding and ψ^{-1} decoding.

By computable we mean we can describe the process of encoding or decoding the tree to string. This description could be only words that have meaning to people but can also be translated to programming language understandable to computers.

Definition 3 (Code, Representation)**.** String $\psi(T)$ for some tree T we will call code of a tree or string representation of a tree (shortly code, or representation).

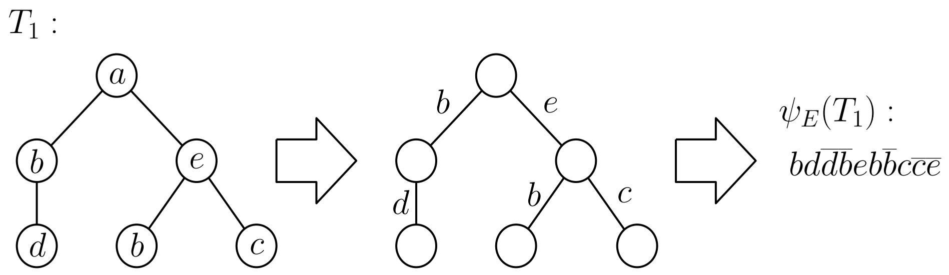
3.1 Euler String

Transformation from a tree to a string is based on the Euler string [3], which is obtained by traversing a tree using the Euler tour. This section contains review of the Euler string coding.

For simplicity, let us treat each tree T as an edge labeled tree: the label of each nonroot node v in the original tree is assigned to the edge $\{u, v\}$ where u is the parent of v. It should be noted that information on the label on the root is lost in this case. But, it is not a problem because the roots are not deleted or inserted. In what follows, we assume that the roots of two input trees have identical labels (otherwise, we just need to add 1 relabel operation to the distance). If we really want to save information about root we can add a new dummy vertex and new edge between this new vertex and the root. So that this edge will have then information about the former root. Label on new dummy root is not important, because it will not be stored in code.

The depth-first search traversal of T (i.e., visiting children of each node according to their left-to-right order) defines an Euler tour of a tree T . That is, the depth-fist search gives an Euler path beginning from the root and ending at the root where each edge $\{w, v\}$ is traversed twice in the opposite directions. Let $\Sigma_S = \{a, \overline{a} | a \in T\}$, where $\overline{a} \notin \Sigma_T$. Let $(e_1, e_2, \ldots, e_{2n-2})$ be the sequence of directed edges in the Euler path of a tree T with n nodes. From this, we create the Euler string $\psi_E(T)$ of length $2n - 2$. Let $e = \{u, v\}$ be an edge in T, where u is the parent of v. Suppose that $e_i = (u, v)$ and

Figure 3.1: Construction of Euler string for tree T_1


T_1 :
a
b
e
d
b
c
b
e
d
b
c
$\psi_E(T_1)$:
$bd\overline{d}\overline{b}eb\overline{b}c\overline{c}\overline{e}$

$e_j = (v, u)$ (clearly, $i < j$). We define $i_1(e)$ and $i_2(e)$ by $i_1(e) = i$ and $i_2(e) = j$, respectively. That is, $i_1(e)$ and $i_2(e)$ denote the first and second positions of e in the Euler tour, respectively. Then, we define $\psi_E(T)$ by letting $\psi_E(T)[i_1(e)] = L(e)$ and $\psi_E(T)[i_2(e)] = \overline{L(e)}$, where $L(e)$ is the label of e (see also Figure 3.1).

Theorem 1. $\psi_E(T_1) = \psi_E(T_2) \quad \Leftrightarrow \quad \tau(T_1, T_2) = 0$. *Moreover, we can reconstruct T from $\psi_E(T)$ in linear time.*

Proof of this theorem and more about Euler string coding can be found in [7, 8].

3.2 Modified Euler String

Modified Euler String has the best results for the worst case scenario. Because if $h = n$, then for all other codings the right side of the boundary inequation the multiplier is n. But for modified Euler string the multiplier is $n^{\frac{3}{4}}$. Although of this qualities, this coding has some serious shortcomings. The biggest shortcoming is that this coding is not a coding function as defined in Definition 2. Because it needs as an argument two trees not only one, and it creates two strings coding these two trees.

Modified Euler String is a modification of Euler String. It works like Euler strings but it modifies some labels on special nodes. The special nodes are vertices that have a small amount of siblings. Small amount is exactly

$\alpha = \sqrt{n} = \sqrt{\max\{|T_1|, |T_2|\}}$. This number is computed from both trees. The boundary is based on the fact that α is not the constant but $\sqrt{n}$. So this function can not create from one tree T_1 its representation. And this function can give different coding for T_1 if the T_2 changes.

For this reasons the modified Euler String is not taken into consideration as much as the other codings.

3.3 Binary Tree Code 1

Every tree can be represented by a binary tree. The Binary Tree Code 1 uses a modified version of a binary tree representation with two kinds of dummy nodes. The Binary Tree Code 1 is a string obtained by traversing the binary tree representation of a tree in preorder.

Definition 4 (binary tree representation 1)**.** Let T be a tree with the root r. Then, a binary tree representation $b_1(T)$ of T is a binary tree obtained by setting, for $v \in T - \{r\}$, the first child of v in T (or $\perp$ if there does not exist) as the left child of v in $b_1(T)$ and the next sibling of v in T (or $\top$ if there does not exist) as the right child of v in $b_1(T)$. In particular, if v is the root r of T, then r is also the root of $b_1(T)$ and left child is the first child of v in T and right child is $\top$.

For a tree T, the string of nodes in T obtained by traversing in preorder denotes $c(T)$.

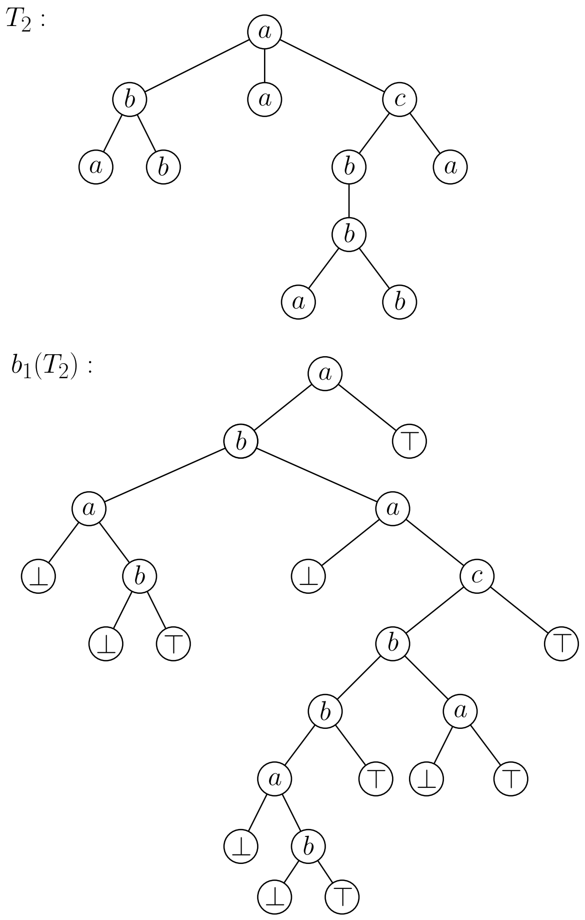
Definition 5 (binary tree code 1)**.** Let T be a tree. Then, the binary tree code 1 of T is the string $c(b_1(T))$, and we denote it by $\psi_{bc1}(T)$.

Example 1. Consider trees T_2 and T_3 in Figure 3.2 (upper). Then, the binary tree representation of T_i is described as Figure 3.2 (lower). Hence, we obtain the binary tree code $\psi_{bc1}(T_i)$ of T_i as follows:

$\psi_{bc1}(T_2) = aba\perp b\perp\top a\perp cbba\perp b\perp\top\top a\perp\top\top\top$

$\psi_{bc1}(T_3) = aaba\perp b\perp\top\top ca\perp bba\perp b\perp\top\top a\perp\top\top\top$

Figure 3.2: A Tree T_i (upper) and a binary tree representation $b_1(T_i)$ of T_i (lower) for $i = 2, 3$



3.4 Binary Tree Code 2

Binary Tree Code 2 uses binary tree representation also as Binary Tree Code 1. But instead of dummy symbols the information about nodes without left or right child is stored in the label of the particular vertex.

Definition 6 (binary tree representation 2)**.** Let T be a tree with the root r. Then, a binary tree representation $b_2(T)$ of T is a binary tree obtained by setting, for $v \in T - \{r\}$, the first child of v in T as the the left child of v in $b_2(T)$ and the next sibling of v in T as the right child of v in $b_2(T)$. In particular if v is the root r of T, then r is also the root of $b_2(T)$. Let $\Sigma_{b_2(T)} = \{a, \overline{a}, \underline{a}, \overline{\underline{a}} | a \in \Sigma_T\}$ where $\overline{a}, \underline{a}, \overline{\underline{a}} \notin \Sigma_T$. For each node we will change label from a to

- $\underline{a}$ - if node does not have left child in $b_2(T)$
- $\overline{a}$ - if node does not have right child in $b_2(T)$
- $\overline{\underline{a}}$ - if node does not have any child in $b_2(T)$

Definition 7 (binary tree code 2)**.** Let T be a tree. Then, the binary tree code 2 of T is the string $c(b_2(T))$, and we denote it by $\psi_{bc2}(T)$.

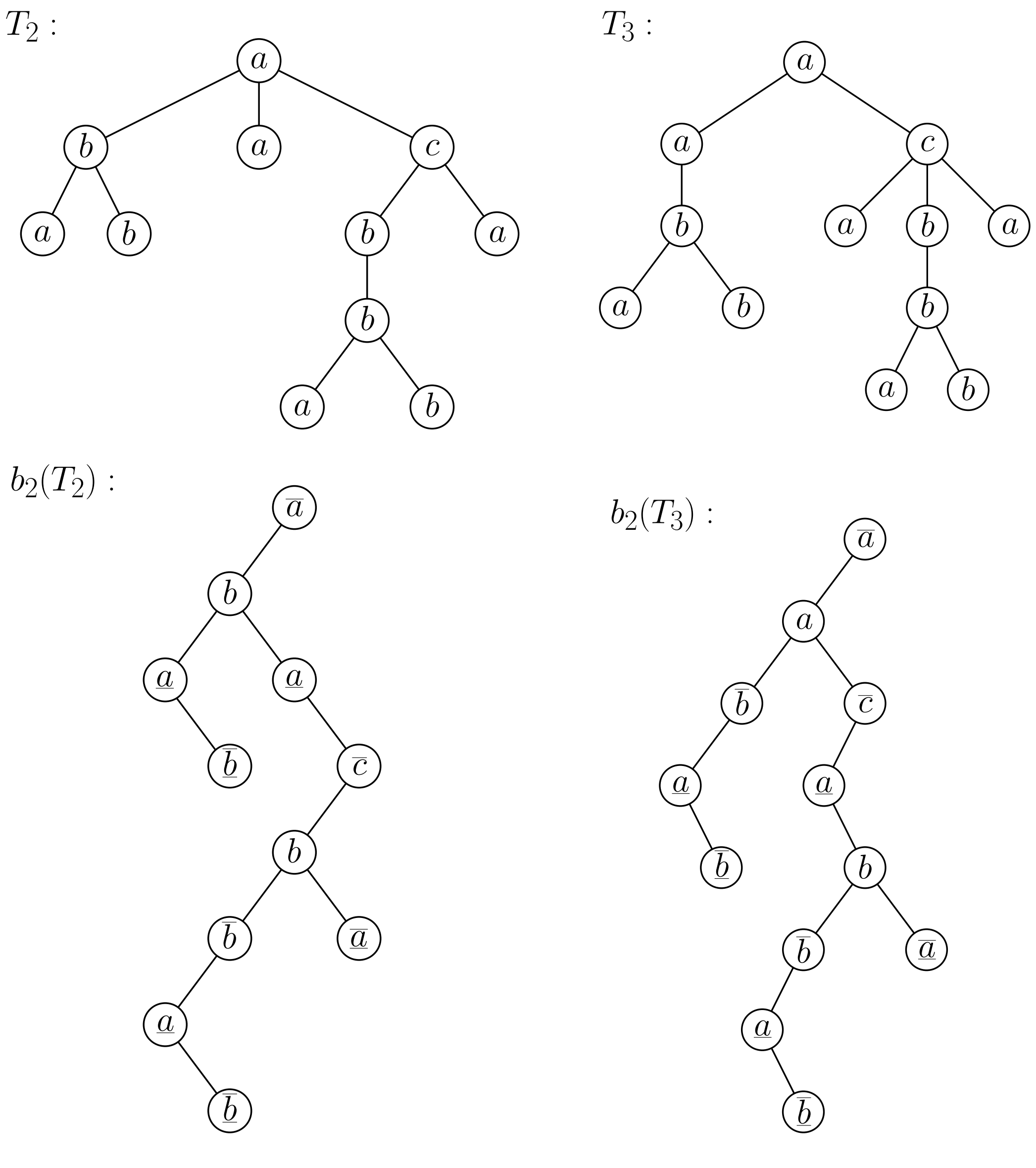
Example 2. Consider trees T_2 and T_3 in Figure 3.3 (upper). Then, the binary tree representation of T_i is described as Figure 3.3 (lower). Hence, we obtain the binary tree code $b_2(T_i)$ of T_i as follows:

$\psi_{bc2}(T_2) = \overline{a}b\underline{a\overline{b}a}\overline{c}b\overline{b}\underline{a\overline{ba}}$

$\psi_{bc2}(T_3) = \overline{a}a\overline{b}\underline{a\overline{b}}\overline{c}\underline{a}b\overline{b}\underline{a\overline{ba}}$

It is obvious that the Binary tree code 1 and the Binary Tree code 2 are very similar. But the Binary tree code 2 has shorter code. At Figure 3.2 and Figure 3.3 are the trees named T_2 and T_3 the same. But the length of the Binary tree code 2 is half of the Binary tree code 1. But the Binary tree code 2 must extend the alphabet Σ_S more. In Binary tree code 2 there is

Figure 3.3: A Tree T_i (upper) and a binary tree representation $b_2(T_i)$ of T_i (lower) for $i = 2, 3$



overlined and underlined letter for every label from Σ_T. But for Binary tree code 1 we just add two dummy letters.

The biggest difference can by found in the Table 3.1. Here we see that the upper bound is multiplied by 2. This reflects the fact that the Binary tree code 2 is shorter and the distance between two code strings gives us an estimate on the distance of the corresponding trees which is twice worse then the estimate obtained vie the Binary tree code 1. The Binary tree code 1 was introduced in [12] during the time we worked on this thesis. The Binary tree code 2 we introduced has similar properties and illustrates the impact of little changes on the result.

3.5 Lower and Upper Bounds for Binary Tree Code 2

We shall now present several Lemmas and Definitions necessary for the proofs of two Theorems that describe relation of $\tau(T_1, T_2)$ and $\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2))$ mentioned in the table in the introduction to this chapter.

Lemma 3. *It holds that* $|b_2(T)| = |\psi_{bc}(T)| = |T|$.

Proof. It is straightforward. □

Lemma 4. *The binary tree code* $\psi_{bc2}(T)$ *can be constructed from a tree* T *in* $O(|T|)$ *time. Conversely, the tree* T *can be constructed from a binary tree code* $\psi_{bc2}(T)$ *in* $O(|T|)$ *time.*

Proof. The first statement is obvious.

For the proof of the second statement, let us use the Binary Tree Code 1. This Lemma is in [12] proved for Binary Tree Code 1. So it is only necessary to prove that Binary Tree Code 1 can be constructed from Binary Tree Code 2 in $O(|T|)$ time.

Putting $\perp$ after every underlined symbol and $\top$ after left subtrees of overlined symbol will construct Binary Tree Code 1. □

Lemma 5. *For trees T_1 and T_2, it holds that $\tau(T_1, T_2) = 0 \Leftrightarrow \psi_{bc2}(T_1) = \psi_{bc2}(T_2)$.*

Proof. By the definition of $\psi_{bc2}(T)$ and by the proof of Lemma 4, it holds that $\psi_{bc2}(T_1) = \psi_{bc2}(T_2) \Leftrightarrow T_1 = T_2$, so the statement holds. $\square$

Lemma 6. *Let $\sqsubseteq$ be a substring relation (overlined, underlined and simple symbols are recognized as the same in $\sqsubseteq$). For a binary tree code $\psi_{bc2}(T)$, we can decode a subtree of T from $\psi_{bc2}(T)$ inductively as follows.*

1. *$\underline{a} \sqsubseteq \psi_{bc2}(T)$ is a subtree of T. In this case, a is a leaf of T.*

2. *If $s_1, \ldots, s_k \sqsubseteq \psi_{bc2}(T)$ are subtrees in T , then $as_1 \ldots s_i \ldots s_k \sqsubseteq \psi_{bc2}(T)$ is a subtree of T. Where i is the most right while the substring relation is true.*

Proof. Let T be a tree and $\psi_{bc2}(T)$ a binary tree code of T . Also let Σ be the set of all labels of nodes in T. By construction of Binary Tree Code 2, underline represents some node v in T has no children, so the statement 1 holds. On the other hand, overline represents some node v in T has no right siblings, so the parent v' of v in T is the root of some subtree of T. Since $\psi_{bc}(T)$ is the preorder traversal of $b_2(T)$, v' is corresponding to the nearest left symbol in $\psi_{bc2}(T)$, so the statement 2 holds. $\square$

Theorem 2. *It holds that $\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2)) \leq \tau(T_1, T_2)$.*

Proof. It is sufficient to show that $\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2))$ changes by at most 2 when an edit operation is applied.

(1) Substitution: Suppose that T_2 is obtained from T_1 by changing a label $u \in T_1$ into a label $v \in T_2$. Then, it is obvious that $\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2)) = 1$.

(2) Deletion: Suppose that T_2 is obtained from T_1 by deleting a node $v \in T_1$, where the root of T_1 is r. Let v_0 be the parent of v in T_1 and $v_1, \ldots, v_n$ the children of v in T_1 . Deleting v in T_1, will cause deletion of v in code

$\psi_{bc2}(T_1)$ and if the node v_0 has children that are right siblings of v, this will be the right children of v_n that has no right child and must change from overlined to be without overline.

So at most 2 changes must be done in binary tree codes. $\square$

For the upper bound on $\tau(T_1, T_2)$ for $\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2))$ let us define similar notions given in [7, 12]. Let $(\psi_{bc2}(T_1)', \psi_{bc2}(T_2)')$ be an alignment giving from $\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2))$. Also let MSP and MSSP be the following sets of pairs obtained from the alignment $(\psi_{bc2}(T_1)', \psi_{bc2}(T_2)')$.

1. MSP is the set $\{(p_1^1, p_1^2), \ldots, (p_d^1, p_d^2)\}$ of maximal substring pairs, each of which is corresponding to a maximal consecutive region in $(\psi_{bc2}(T_1)', \psi_{bc2}(T_2)')$ without insertions, deletions or substitutions.
2. $MSSP$ is the set $\{(t_1^1, t_1^2), \ldots, (t_b^1, t_b^2)\}$ of maximal subtree string pairs, each of which is corresponding to a maximal subtree of T_i by decoding $\psi_{bc2}(T_i)$ in Lemma 6.

Note that $MSSP$ is determined uniquely from the alignment $(\psi_{bc2}(T_1)', \psi_{bc2}(T_2)')$. For an $MSSP$, we construct the mapping $\mathcal{M}$ from the nodes in t_i^1 to ones in t_i^2 (with ignoring underline and overline).

Lemma 7. *$\mathcal{M}$ is a bottom-up mapping between T_1 and T_2.*

Proof. It is straightforward. $\square$

Lemma 8. *If $|MSP| = d$, then it holds that $d \leq \delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2)) + 1$.*

Proof. Let MSP be $\{(p_1^1, p_1^2), \ldots, (p_d^1, p_d^2)\}$. Then, there exist at least $d-1$ gaps in the alignment $(\psi_{bc2}(T_1)', \psi_{bc2}(T_2)')$. Hence, it holds that $d - 1 \leq \delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2))$, so the statement holds. $\square$

Let $REST(p_i^j)$ be the total number of positions in substring p_i^j that do not appear in $MSSP$. Also let h be the minimum height of T_1 and T_2.

Lemma 9. *For every p_i^j, it holds that $REST(p_i^j) \leq h$.*

Proof. Let P_{h+1} be a path $a_0 \dots a_h$ with height h ($h+1$ nodes), where the root is a_0. Then, it holds that $bc'(P_{h+1}) = a_0 a_1 \dots a_{h-1} a_h$. Consider the case that $p_i^j = a_0 a_1 \dots a_{h-1} a_h$ and $t_i^l = a_h$ (for some $1 \leq l \leq b$) Then, no substring of p_i^j except t_i^l is decoded as a subtree of P_{h+1} , so it holds that $REST(p_i^j) = h$, which is the worst case for $REST(p_i^j)$. □

Lemma 10. *Let k be $\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2))$. Then, it holds that*

$$\sum_{j=1}^{d} |p_i^j| \geq |T_i| - k$$

Proof. By Lemma 3, it holds that $|\psi_{bc2}(T_i)| = |T_i|$ and the length of $\psi_{bc2}(T_i)$ is the sum of $|\psi_{bc2}(T_i)|$ and the number of gaps. Since $k(= \delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2)))$ is greater than the number of gaps, which is $|T_i| - \sum_{j=1}^{d} |p_i^j|$ the statement holds. □

Theorem 3. *Let h be the minimum height of T_1 and T_2. Then, it holds that $\tau(T_1, T_2) \leq 2(h+1)\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2)) + 2h$.*

Proof. Let $\mathcal{M}$ be a bottom-up mapping as in Lemma 7. Then, $\mathcal{M}$ is corresponding to the elements of t_1^l and t_2^l. Also let k be $\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2))$.

For $p_i^j (1 \leq j \leq d)$, the number of positions assigned by $\mathcal{M}$ is at least $|p_i^j| - REST(p_i^j)$, and also at least $|p_i^j| - h$ by Lemma 9. Then, the following sequence holds for $i = 1, 2$.

$$|M| \geq \sum_{j=1}^{d} (|p_i^j| - h) = \sum_{j=1}^{d} |p_i^j| - dh$$

$$|M| \geq \sum_{j=1}^{d} |p_i^j| - h(k+1) \quad \text{(by Lemma 8)}$$

$$|M| \geq |T_i| - k - h(k+1) \quad \text{(by Lemma 10)}$$

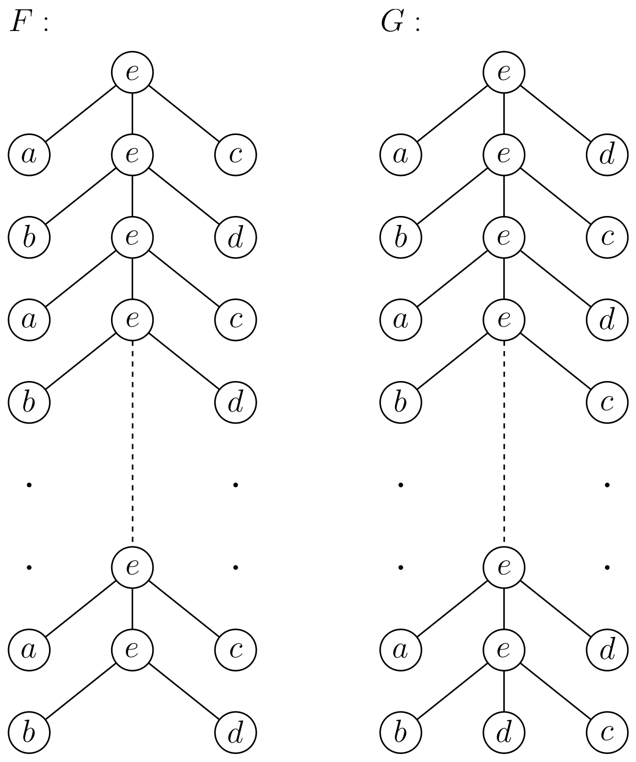
$$|M| \geq |T_i| - k(h+1) - h$$

Hence, the following sequence holds.

$$\begin{aligned} \tau(T_1, T_2) &\leq |T_1| + |T_2| - |M| - id(M) = |T_1| + |T_2| - 2|M| \\ &\leq |T_1| + |T_2| - (|T_1| + |T_2| - 2k(h+1) - 2h) \\ &= 2k(h+1) + 2h. \end{aligned}$$

So $\tau(T_1, T_2) \leq 2(h+1)\delta(\psi_{bc2}(T_1), \psi_{bc2}(T_2)) + 2h$. □

Figure 3.4: The worst case of tree sets for tree distance and string distance of their representation



3.6 Upper Bound is tight

For Euler String and Binary Tree Codes 1 and 2 the upper bound shown in Table 3.1 is tight. The proved upper bounds for this codings can not be better. The following example shows the worst case when the string distance of tree codes for the particular coding is almost equal to the upper bound.

Example 3. Trees on Figure 3.4 have tree edit distance equal to h. Because we must change the label of at least one vertex for every level. But string distances of tree encodings is constant. Let us look at these encodings.

Euler String:
$\psi_E(F) = (ea\overline{a}eb\overline{b})^m(d\overline{d}\overline{e}c\overline{c}\overline{e})^m \quad = (ea\overline{a}eb\overline{b})^m d\overline{d}\overline{e}(c\overline{c}\overline{e}d\overline{d}\overline{e})^{m-1}c\overline{c}\overline{e}$
$\psi_E(G) = (ea\overline{a}eb\overline{b})^m d\overline{d}(c\overline{c}\overline{e}d\overline{d}\overline{e})^m = (ea\overline{a}eb\overline{b})^m d\overline{d}\ \ (c\overline{c}\overline{e}d\overline{d}\overline{e})^{m-1}c\overline{c}\overline{e}d\overline{d}\overline{e}$

String distance of these codes is $\delta(\psi_E(F), \psi_E(G)) = 4$.

Binary Tree Code 1:

Obvious codes from the figure:
$\psi_{bc1}(F) = (ea\bot eb\bot)^m(d\bot\top c\bot\top)^m\top$
$\psi_{bc1}(G) = (ea\bot eb\bot)^m d\bot(c\bot\top d\bot\top)^m\top$

Aligned codes for better distance computing:
$\psi_{bc1}(F) = (ea\bot eb\bot)^m d\bot\top(c\bot\top d\bot\top)^{m-1}c\bot\top \qquad \top$
$\psi_{bc1}(G) = (ea\bot eb\bot)^m d\bot \quad (c\bot\top d\bot\top)^{m-1}c\bot\top d\bot\top\top$

String distance of these codes is $\delta(\psi_{bc1}(F), \psi_{bc1}(G)) = 4$ too.

Binary Tree Code 2:

$\psi_{bc2}(F) = \overline{e}\underline{a}e\underline{b}(e\underline{a}e\underline{b})^{m-1}(\underline{\overline{d}\overline{c}})^m = \overline{e}\underline{a}e\underline{b}(e\underline{a}e\underline{b})^{m-1}\underline{\overline{d}}(\underline{\overline{c}\overline{d}})^{m-1}\underline{\overline{c}}$
$\psi_{bc2}(G) = \overline{e}\underline{a}e\underline{b}(e\underline{a}e\underline{b})^{m-1}\underline{d}(\underline{\overline{c}\overline{d}})^m = \overline{e}\underline{a}e\underline{b}(e\underline{a}e\underline{b})^{m-1}\underline{d}(\underline{\overline{c}\overline{d}})^{m-1}\underline{\overline{c}\overline{d}}$

String distance for this encoding is $\delta(\psi_{bc2}(F), \psi_{bc2}(G)) = 2$. We just need to overline d and insert one d at the end.

String distances for these codes are always constant independent from variable m. But height of the tree is dependent on m. So we have two set of trees from which we can always choose two, one from each set, for that string distance of tree codes will be constant and tree distance will be minimal height of trees. And the height of trees we can have whatever we want. So for these trees F and G this is true:

$$\tau(F, G) = \Theta(h) \cdot \delta(\psi(F), \psi(G))$$

Chapter 4

Relating tree and string distances

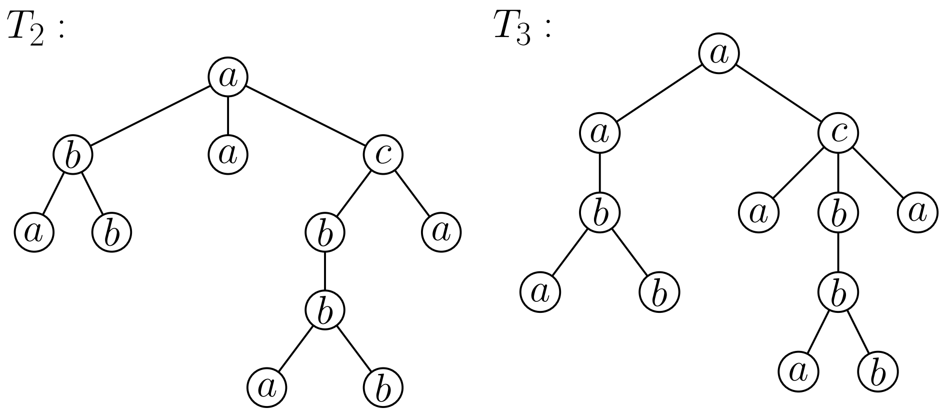
Relation between tree distance and string distance of their codes we discussed already in the chapter before. However in chapter before we discussed this relation for particular codings. In this chapter we will discuss this relation in general.

This chapter contains main proof of this thesis that for codings with some nice properties we can not get rid of h in the upper bound of distances. But where do we get these nice properties? This is discussed before the main proof. First section contains coding with very bad properties that result in bad bounds between distances. Although these coding have bad results we can learn what circumstances we must try to avoid in the definition of nice properties for coding in general. Next section contains definitions of nice properties with motivations for them and short descriptions what they give us. And end of this chapter is of course given to the proof.

4.1 Enlightenment from bad case

Sometimes we can learn from something bad. Here is such an example of this case. In my research I discovered a coding with very bad boundaries.

Figure 4.1: Example of some trees to show Level Code



Although we can say this is bad coding, there is something we can learn from it. This coding helps me to discover the requirements needed to prove the Theorem 4.

Following content of this section contains description of this bad coding we will call Level Code. We shall provide examples demonstrating the undesirable behavior of this coding. This coding helped us to identify "bad features" we would like to avoid in "good codes". Despite these bad properties the Level Code overcomes one of the problems encountered in codes mentioned in Chapter 3, namely dependence of the worst case error on the height of the trees.

Definition 8 (Level Code)**.** Level Code we will construct from tree T like this. We will store vertices as they are in the levels. The first level is the root. The second level is its children. The third level is their children. The order we can get from the tree because it is an ordered tree. Next we must put some extra information so we can reconstruct the tree from its code. For vertices without children we can use underline. For vertices that is the last child of its parent we will use overline. Root will be overlined too so we can easily represent forest. Notation for function creating level code is ψ_{lc}.

Example 4. Level codes for trees shown in Figure 4.1 are the following:

$\psi_{lc}(T_2) = \overline{a}b\underline{a}\overline{c}\underline{a\overline{b}}b\underline{\overline{a}}\overline{b}\underline{a\overline{b}}$
$\psi_{lc}(T_3) = \overline{a}a\overline{c}\overline{b}\underline{a}b\underline{\overline{a}a\overline{b}}\overline{b}\underline{a\overline{b}}$

Example 5 (Bad case for others, not for Level Code)**.** Now let us consider trees from Figure 3.4 that represent the worst case for other codings because string distance of codes was constant. Level code of these trees looks like this:
$\psi_{lc}(F) = \overline{e}\underline{a}e\underline{\overline{c}b}e\underline{\overline{d}a}e\underline{\overline{c}b}e\underline{\overline{d}}\ldots\underline{a}e\underline{\overline{c}b\overline{d}}$
$\psi_{lc}(G) = \overline{e}\underline{a}e\underline{\overline{d}b}e\underline{\overline{c}a}e\underline{\overline{d}b}e\underline{\overline{c}}\ldots\underline{a}e\underline{\overline{d}bd\overline{c}}$
It is obvious that $\delta(\psi_{lc}(F), \psi_{lc}(G)) = h$. Which is an interesting result.

Now let us look why this coding is mentioned as bad. For now it looks like a good coding. We can encode tree into string. We can decode tree from a correct code of a tree. String distance of the worst case for other codes is better, because as we have seen it is equal to the tree distance.

Example 6 (Lower bound of Level Code)**.** Consider trees T_4, T_5 at Figure 4.2. Tree T_5 we get from T_4 by removing vertex with label a_2. One elementary operation to get one tree from other means $\tau(T_4, T_5) = 1$. Let us look at the Level Codes of these trees.
$\psi_{lc}(T_4) = \overline{a}a_1\overline{a_2a_3a_4a_5a_6a_7a_8}\ldots$
$\psi_{lc}(T_5) = \overline{a}a_1\overline{a_4a_3a_6a_5a_8a_7}\ldots$
Distance between these two trees is dependent on h. It is also $\frac{n}{2}$ because in this case $h = \frac{n}{2}$.

Trees T_6, T_7 on Figure 4.2 shows that h is not significant. These trees have constant height. Tree distance is too 1 because T_7 we get from T_6 by removing vertex with label a_2. Level Codes of these trees:
$\psi_{lc}(T_6) = \overline{c}c_1\overline{c_2}\underline{c_3c_4}\ldots\underline{c_{\frac{n}{2}}}\ldots$
$\psi_{lc}(T_7) = \overline{c}c_1\underline{c_{\frac{n}{2}}}\ldots\underline{c_3c_4}\ldots$
It is obvious that $\delta(\psi_{lc}(T_6), \psi_{lc}(T_7)) = n - 2$. First two letters are the same. To change the other we can rename $n-2$ characters, or delete and insert half of the vertices. Either way we use the same number of elementary operations.

Figure 4.2: Trees showing bad lower bound for Level Code

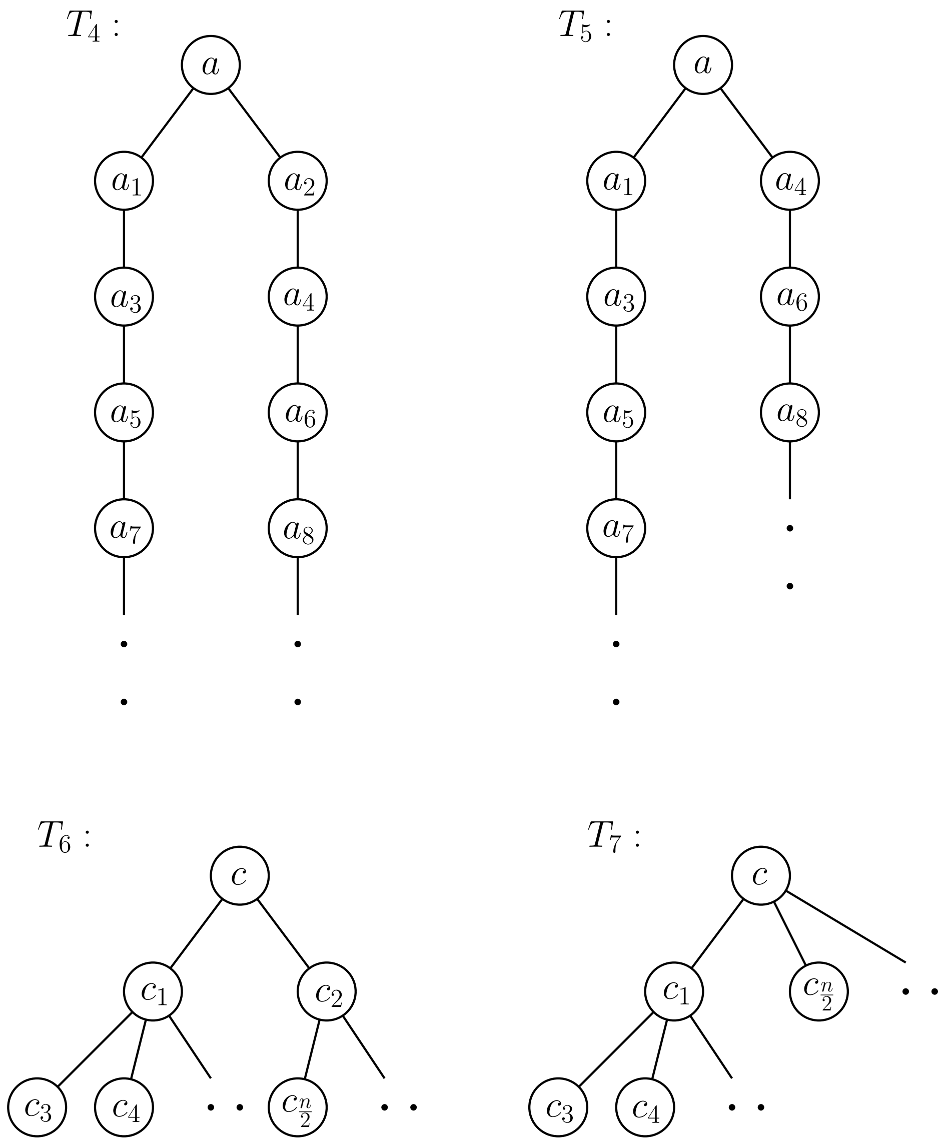
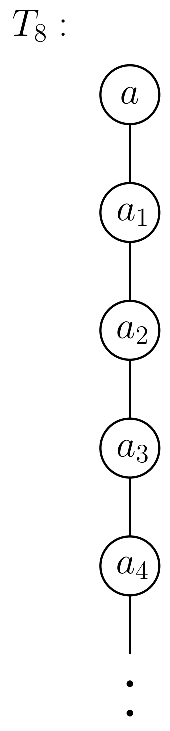


Figure 4.3: Tree T_8 showing bad lower bound for Level Code



From this example we see that lower bound for Level Code is:

$$\frac{1}{n}\delta(\psi_{lc}(F), \psi_{lc}(G)) \leq \tau(F, G)$$

This is really a bad result. The codings before has a constant multiplier instead of n like here. For codings before elementary operations on trees causes constant change for the code. In Level Code the change could be up to almost the number of vertices.

Now we will focus on the upper bound. The following example shows this coding is worse them codings described in the previous chapter in the upper bound too.

Example 7 (Upper bound of Level Code)**.** We will use tree T_4 from lower bound example at Figure 4.2 and compare it with tree T_8 shown at Figure 4.3. Let us see their codes.
$\psi_{lc}(T_4) = \overline{a}a_1\overline{a_2a_3a_4a_5a_6a_7a_8}\ldots$

$\psi_{lc}(T_8) = \overline{aa_1a_2a_3a_4a_5a_6a_7a_8}\ldots$
The difference in these codes is an overline above a_2 and first code must contain one more underlined letter at the end. So $\delta(\psi_{lc}(L_1), \psi_{lc}(U)) = 2$. But tree distance is depended on n.

Upper bound for Level Code:

$$\tau(F, G) \le n \cdot \delta(\psi_{lc}(F), \psi_{lc}(G))$$

Why bother with such a bad code? This code has much worse boundaries than codes noted before. So why study this code? This encoding gives us a special knowledge. How to describe in general the difference between this code and good codes? Thinking about this difference lead me to Definition 11. This definition avoids coding like this. If we look to the definition the difference is obvious. In Level Coding codes of children of some vertex v is after all nodes in the same level. This means children of node v appear in the code after siblings of node v. In good codes if v is a sibling between vertices w and z children of v are in the code between vertices w and z.

4.2 Requirements

To prove that upper bound is multiplied by $\Omega(h)$ in general, we need to formulate some natural requirements for coding functions to satisfy. These requirements are mentioned here as three definitions. This section also contains motivations and descriptions of ideas that led to these requirements.

First of these requirements is a stability of coding. By defining stable coding we are trying to avoid codings that change order of storing vertices as a result of changing a label of some vertex. All codes mentioned in Chapter 3 satisfy this. The structure of these codings does not depend on labels of vertices.

Definition 9 (Stable coding)**.** Let ψ be a coding as defined in Definition 2. Let F, G be trees with these properties: F has a distinct labels on vertices.

And G is constructible from F by relabeling its vertices. Formally:

$$\exists h : \Sigma_{T_F} \to \Sigma_{T_G} \quad \text{such that} \quad h(F) = G$$

So h is a homomorphism from labels of F to labels of G and G can be constructed applying h to labels of vertices from F.

$$\psi \text{ is stable} \quad \Longleftrightarrow \quad \exists h' : \Sigma_{S_F} \to \Sigma_{S_G} \quad \text{such that}$$

$$h'/\Sigma_{T_F} = h \quad \text{and} \quad h'(\psi(F)) = \psi(G)$$

The second requirement we shall define below is the elementary inversibility of the coding. With this definition we are avoiding codings that are trying to accumulate information to one spot in the code. Elementarily inversibile coding produces code such that we can assign one letter of this code to at most one vertex of tree. It is obvious that all the codes mentioned here satisfy this. A letter of these codes is storing information about at most one vertex of the tree.

Before we define elementarily inversibile coding we need to describe an assignment function a. For every coding ψ we can define an assignment function for positions of letters in code $s = \psi(T)$. An assignment function will assign every position in s a set of vertices from T such that relabeling a node from this set will change the letter in that position. The assignment function is denoted by a.

$$a(i) = \{v_1, v_2, \ldots, v_k\} \Longleftrightarrow \forall_{j=1,2,\ldots k} \text{ relabeling of } v_j \text{ changes } s[i]$$

Definition 10 (Elementarily inversibile coding)**.** Coding ψ is elementarily inversibile $\Longleftrightarrow \forall T \quad \forall_{i=1,2,\ldots,length(s)} |a(i)| \leq 1$

The third requirement is the projecting tree structure feature. Motivation and ideas leading to this property is described in the previous section. We are trying to force the coding to store vertex and its offspring between the code of its siblings.

We will use inversion of the assignment function a. It will be used to identify all positions in the code which are influenced by a particular vertex of a tree.

$$a^{-1}(v) = \{i_1, i_2, \ldots, i_k\} \text{ such that } \forall_{l=1,2,\ldots,k} s[i_l] \text{ is code of } v \text{ in } \psi(T)$$

We shall now define a property of codings which ensures that the tree structure is suitably represented by subword structure. Namely, all symbols related to a particular vertex (and its descendants) occuring in the code appear between those of its siblings.

Definition 11 (Projecting tree structure)**.** Let ψ be a coding. Let T be a tree and vertices $v, w, x \in T$ be siblings ordered v, w, x. Let $T(w)$ be a vertices of T_w, a subtree of T rooted at w. Coding ψ is projecting tree structure when

$$\forall i \in a^{-1}(v) \quad \forall k \in a^{-1}(x) \quad \forall z \in T(w) \quad \forall j \in a^{-1}(z) \qquad i \leq j \leq k$$

or

$$\forall i \in a^{-1}(v) \quad \forall k \in a^{-1}(x) \quad \forall z \in T(w) \quad \forall j \in a^{-1}(z) \qquad i \geq j \geq k$$

holds.

In other words if ψ is projecting tree structure then the code can be divided like this $\psi(T) = s_1 \cdot S_1 \cdot s_2 \cdot S_2 \cdot s_3$ where S_1, S_2 are letters coding vertices v, x and letters coding vertex w and its children are contained only in string s_2.

For elementary inversibile coding holds that for two different vertices v, w is $a^{-1}(v) \cap a^{-1}(w) = \emptyset$. In detail $\forall_{i \in a^{-1}(v)} \forall_{j \in a^{-1}(w)} \quad i \neq j$. If coding ψ is elementary inversibile and projecting tree structure then it holds

$$\forall i \in a^{-1}(v) \quad \forall k \in a^{-1}(x) \quad \forall z \in T(w) \quad \forall j \in a^{-1}(z) \qquad i < j < k$$

or

$$\forall i \in a^{-1}(v) \quad \forall k \in a^{-1}(x) \quad \forall z \in T(w) \quad \forall j \in a^{-1}(z) \qquad i > j > k$$

These properties give us powerful tools for analyzing codings that is useful for us in general.

4.3 General Proof

Everything is set to begin the proof. First we prove Lemma 11 that gives us better and more usable tool for stable property which we will use later in the proof of Theorem 4.

Lemma 11. *Let ψ be a stable coding function. Let T, F, G be a trees, where T has distinct labels on vertices and F, G can be constructed from T by relabeling vertices. Let f, g be the homomorphisms that construct F, G from T. If*

$$\forall a \in \Sigma_F \quad \exists b \in \Sigma_G \qquad f^{-1}(a) \subseteq g^{-1}(b)$$

then there exist homomorphisms h, h' such that $h(F) = G$, and $h'(\psi(F)) = \psi(G)$ hold.

Proof. We must construct h and h' and show that $h \cdot f = g$ and $h' \cdot f' = g'$.

First h. Because $f(T) = F$ and $g(T) = G$ so if $h \cdot f = g$ than it means that $h \cdot f(T) = h(f(T)) = h(F) = G = g(T)$. So we will construct h like this:
$h = g \cdot f^{-1}$ in detail $h(a) = g(f^{-1}(a))$.

$f^{-1}(a)$ is a set. But preposition of this lemma says that $\forall_{a \in F} \exists_{b \in G} \quad f^{-1}(a) \subseteq g^{-1}(b)$. So for every $a \in \Sigma_F$ we get $b \in \Sigma_G$ for that $g(f^{-1}(a)) = b$. So h is a properly defined homomorphism.

Now h'. From stability we got homomorphisms f', g' that are extensions of homomorphisms f, g for the same coding ψ. This means that the proposition will hold for extensions too, and we could construct h' from f', g' as we did h from f, g. $\square$

Theorem 4. *Let ψ be a coding function with these properties:*

- *ψ is a stable coding function (Definition 9)*
- *ψ is an elementarily inversibile coding (Definition 10)*
- *ψ is projecting tree structure (Definition 11)*

- *$\exists$ constant k $\forall$ trees T, U* $\quad \delta(\psi(T), \psi(U)) \leq k \cdot \tau(T, U)$

then there exist sets of trees $\mathcal{F}, \mathcal{G}$ such that

$$\forall F \in \mathcal{F} \quad \forall G \in \mathcal{G} \qquad \tau(F, G) = \Omega(h) \cdot \delta(\psi(F), \psi(G))$$

where h is the minimal height of trees F, G.

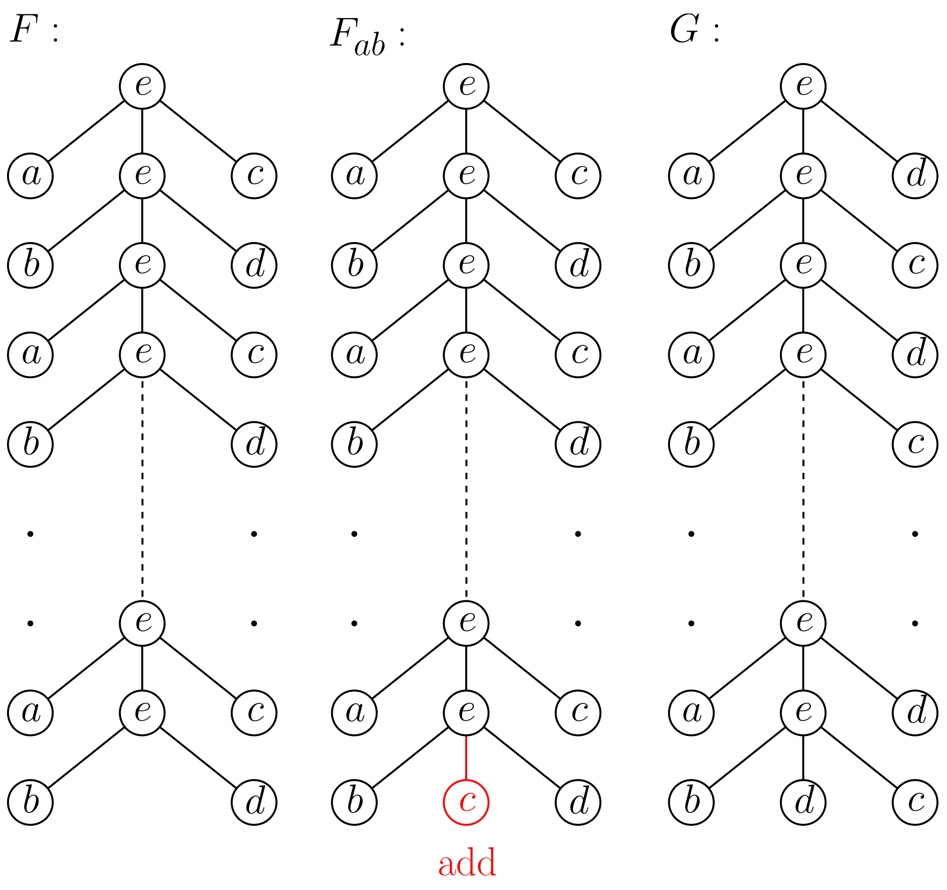
Proof. The same tree sets for the generalized proof will be used that were used earlier for concrete codings. These pictures are shown at Figure 3.4.

Let us think about how $\psi(F)$ and $\psi(G)$ will look. They could not be described exactly, but only what we can say from F, G and properties of coding ψ.
$\psi(F)$: From tree structure projecting property we know that there must be coding of the top vertices labelled a and c and between them must be their brother vertex labelled e and all its children. Although we do not know the exact a and c order in the $\psi(F)$, we can suppose that the order is a in the left and c in the right side of the code. This is thanks to the stability of the code. Because if the coding gives us the opposite coding we can switch the label between a and c and stability gives us the same code with the switched order of these two vertices. And the proof will be the same. The tree sets will be different, but the codes will have the same order.
Formally $\psi(F)$ will look like this:

$\psi(F) = a_1 \cdot A_1 \cdot z_1 \cdot C_1 \cdot c_1$ where a_1, z_1, c_1 are strings
A_1, C_1 are codes of vertices labeled: a, c

We can continue to properly describe z_1 so the second thought will look like this:
$\psi(F) = a_1 \cdot A_1 \cdot b_2 \cdot B_2 \cdot z_2 \cdot D_2 \cdot d_2 \cdot C_1 \cdot c_1$
And we can continue like this.
$\psi(G)$: The code will look like this: $G_{ab} \cdot D_h \cdot G_{cd}$
Starting G_{ab} means code that contains no vertices labelled c or d. Similarly G_{cd} means code that does not contain vertices labelled a or b.

Figure 4.4: Construction of F_{ab}



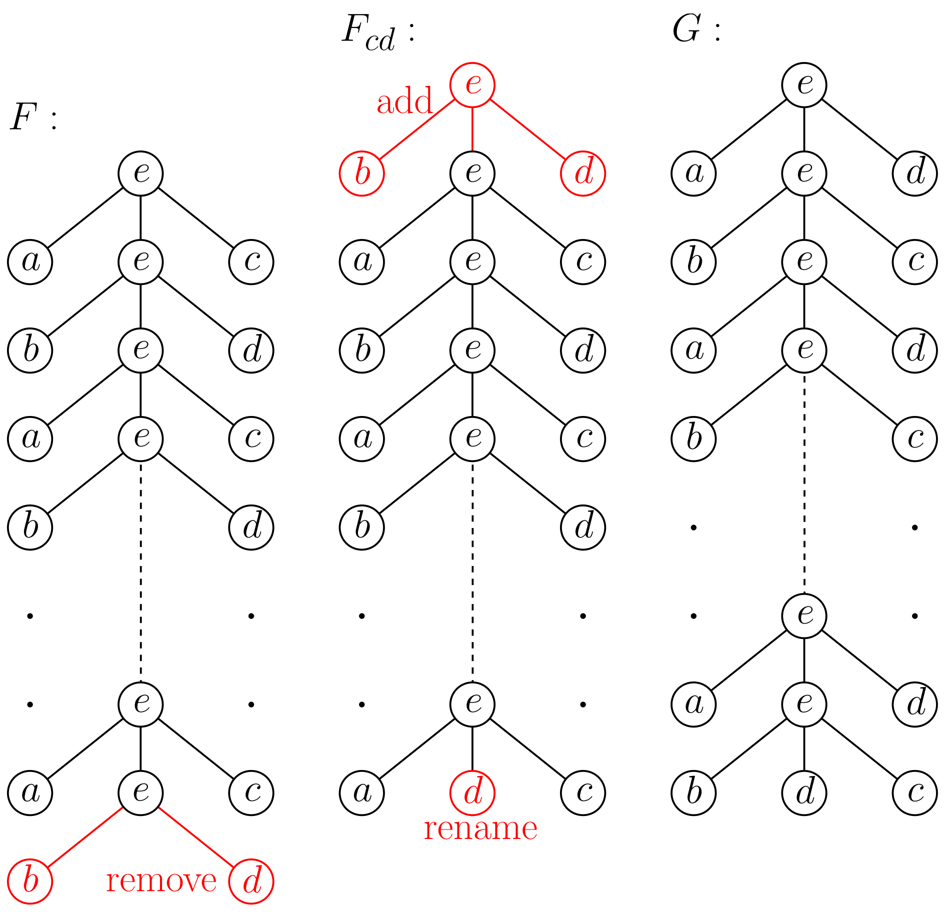
Let us now look how will look the code of the tree F_{ab} that can be constructed from F by one adding operation. Tree F_{ab} is shown on Figure 4.4. $\tau(F, F_{ab}) = 1$ because $F \neq F_{ab}$ and F_{ab} is constructable by one adding operation from F.

$\delta(\psi(F), \psi(F_{ab}) \leq k_1\tau(F, F_{ab}) = k_1 \cdot 1 = k_1$ for some constant k_1

$\psi(F_{ab})$: comparing F_{ab} with G that there exist homomorphism h_1 for that $F_{ab} = h_1(G)$, because h_1 could be simply defined like this $h_1(c) = d$,$h_1(d) = c$ and for other labels it is identical. Using the Lemma 11 we can say that there exist homomorphism h'_1 extended from h_1 that affects the coding. So for h'_1 it holds that $\psi(F_{ab}) = h'_1(\psi(G))$

$\psi(F_{ab}) = h'_1(\psi(G)) = h'_1(G_{ab} \cdot D_h \cdot G_{cd}) = G_{ab} \cdot h'_1(D_h \cdot G_{cd})$

Figure 4.5: Construction of F_{cd}



G_{ab} is not affected by h'_1 because it does not contain codings of labels c or d. And for the labels other than codes of c or d the h_1 is defined as identical homomorphism and also the h'_1 will not affect the code of these vertices.

Now let us think about tree F_{cd} that is shown on Figure 4.5. F_{cd} can be constructed from F by 6 elementary edit operations: 2 remove, 1 relabel and 3 add operations. So $\tau(F, F_{cd}) \leq 6$
$\delta(\psi(F), \psi(F_{cd})) \leq k_2 \cdot \tau(F, F_{cd}) \leq k_2 \cdot 6 = 6k_2$ for some constant k_2

Using the same reasoning as for F_{ab} we get:
$\psi(F_{cd}) = h'_2(G_{ab}) \cdot D_h \cdot G_{cd}$
where h_2 is homomorphism such that $h_2(a) = b$ and $h_2(b) = a$ and it is identity for other labels. And h'_2 is extended homomorphism from homomor-

phism h_2 from Lemma 11. So we have
$\psi(F_{ab}) = G_{ab} \cdot h'_1(D_h \cdot G_{cd})$ where $\delta(\psi(F), \psi(F_{ab})) \leq k_1$
$\psi(F_{cd}) = h'_2(G_{ab}) \cdot D_h \cdot G_{cd}$ where $\delta(\psi(F), \psi(F_{cd})) \leq 6k_2$
Let $\psi(F) = F_1 \cdot F_2$ where
F_1 - minimal part of the $\psi(F)$ code that contains all the vertices that are labelled a or b
F_2 - rest of the code $\delta(\psi(F_1), \psi(G_{ab})) \leq k_1 + k_3 = K_1$ where K_1 is constant
c_3 - constant needed to remove rest of the vertices labelled c or d from the right side that can be mixed with vertices labelled a or b. Constant one could be mixed because for higher level the tree structure projective property can be used.
$\delta(\psi(F_2), \psi(D_h \cdot G_{cd})) \leq 6k_2 + k_4 = K_2$ where K_2 is constant
c_4 - constant needed to add rest of the last vertices labelled c or d that were in F_1. To sum it up:
$\delta(\psi(F), \psi(G)) \leq k_1 + 6k_2 + k_3 + k_4 = K_1 + K_2 =$ constant
$\tau(F, G) = h + 1$

$\square$

Chapter 5

Conclusions

It seems that comparing representations of trees does not give us exact information about tree distance. Proof in this thesis shows us that ideal coding for trees probably does not exist. We say probably because we had very strong requirements for coding to prove this result. Maybe someone will find coding that does not have all this properties and it will be good coding for him. Because it has features we want.

One way is definition of our coding. The Modified Euler String is a function that does not fulfil our definition of coding. As we could see it brings better results than codings that fulfil our definition. Improving of these results could bring some new results.

Another way of studying this area is to take a better look at the requirements mentioned right before the proof. Definitions of properties like stable coding, elementarily inversibile, projecting tree structure, or requirement of lower bound are these requirements. Do we need all of them? Could three of them imply the fourth? Is there a nice encoding we want to study that does not satisfy the properties in these definitions? These are the questions for further study.

Bibliography

[1] TAI, K. 1979. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery (JACM) 26*, 3, 422-433.

[2] SHASHA, D. AND ZHANG, K. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing 18*, 6, 1245-1262.

[3] KLEIN, P. N. 1998. Computing the edit-distance between unrooted ordered trees. *In Proceedings of the 6th annual European Symposium on Algorithms (ESA).* 91-102.

[4] DULUCQ, S. AND TOUZET, H. 2003. Analysis of tree edit distance algorithms. *In Proceedings of the 14th annual symposium on Combinatorial Pattern Matching (CPM).* 83-95.

[5] DEMAINE, E. D., MOZES, S., ROSSMAN, B., AND WEIMANN, O. 2007. An optimal decomposition algorithm for tree edit distance. *In Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP).* 146-157.

[6] HAREL, D. AND TARJAN, R. E. 1984. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing 13*, 2, 338-355.

[7] AKUTSU, T. 2006. A relation between edit distance for ordered trees and edit distance for Euler strings. *Information Processing Letters 100*, 105-109.

[8] Valiente, G. 2002. Algorithms on Trees and Graphs. *Springer, Berlin.*

[9] Grossi, R. 1993. On finding common subtrees. *Theor. Comput. Sci. 108*, 345-256

[10] Akutsu, T., Fukagawa, D., and Takasu, A. 2006. Approximating tree edit distance through string edit distance, *Proc. 17th International Symposium on Algorithms and Computation (ISAAC 2006), Lecture Notes in Computer Science 4288*, 90-99.

[11] Bille, P. 2005. A survey on tree edit distance and related problems. *Theoretical computer science 337*, 217-239.

[12] Aratsu, T., Hirata, K., and Kuboyama, T. 2009. Approximating Tree Edit Distance through String Edit Distance for Binary Tree Codes. *In Proceedings of the 35th Conference on Current Trends in theory and Practice of Computer Science (Spindlerov Mlýn, Czech Republic, January 24 - 30, 2009). M. Nielsen, A. Kučera, P. B. Miltersen, C. Palamidessi, P. Tůma, and F. Valencia, Eds. Lecture Notes In Computer Science, vol. 5404. Springer-Verlag, Berlin, Heidelberg*, 93-104.

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky
Katedra informatiky

Vzdialenosti na stromoch a ich reprezentáciách

(diplomová práca)

David Zachar

Vedúci: prof. RNDr. Branislav Rovan, PhD. Bratislava, 2010

Týmto prehlasujem, že som diplomovú prácu vypracoval samostatne s použitím uvedenej literatúry a s odbornou pomocou diplomového vedúceho.

. . . .
David Zachar

Abstrakt

Cielom tejto práce je preskúmať akú informáciu o rozdiele dvoch stromov môžeme získať z rozdielu ich textových reprezentácií. Najskôr sa zameriame na nápady ako počítať rozdiel dvoch stromov nazývaný vzdialenosť, ktoré boli publikované v iných prácach. Potom prejdeme k jednotlivým textovým reprezentáciám stromov. V tejto práci definujem nové kódovanie. Vzťah medzi stromovou vzdialenosťou dvoch stromov a reťazcovou vzdialenosťou ich reprezentácií získaných týmto kódovaním je viditeľný z dôkazov dolného a horného ohraničenia pre toto kódovanie. Z jednotlivých kódovaní stromov na reťazce to vyzerá tak, že my nemôžeme získať presnú informáciu o stromovej vzdialenosti zo vzdialenosti ich textových reprezentácií. Táto práca obsahuje dôkaz, že pre každé kódovanie ψ, ktoré spĺňa niektoré prirodzené vlastnosti, platí, že nemôžeme získať presnú informáciu o stromovej vzdialenosti zo vzdialenosti kódov kódovaných ψ. Formálne, existujú stromy F a G, pre ktoré platí, že $\tau(F,G) = \Omega(h)\delta(\psi(F),\psi(G))$. Kde τ je stromová vzdialenosť, δ je reťazcová vzdialenosť a h je minimálna výška stromov F a G.

Poďakovanie

Ďakujem vedúcemu mojej diplomovej práce Branislavovi Rovanovi za dôležité vedenie, veľmi užitočné rady a prospešné diskusie počas tvorby tejto práce.