FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Automata Approach to Approximate Tree Pattern Matching
Student:	Bc. Lukáš Renc
Supervisor:	Ing. Eliška Šestáková
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2020/21

Instructions

Study methods for approximate tree pattern matching; see the automata approach for constrained approximate subtree matching introduced in [1]. Based on this research, propose your own method for approximate tree pattern matching that would support more general approximate distances than [1] (e.g., 1-degree edit distance introduced by Selkow in [2]). The proposed method should be based on suitable formal models from the theory of formal languages and automata. Discuss the theoretical time and space complexities of your proposed method and implement it. Perform appropriate testing of your implementation.

References

[1] ŠESTÁKOVÁ, Eliška; MELICHAR, Borivoj; JANOUŠEK, Jan. Constrained Approximate Subtree Matching by Finite Automata. In: Prague Stringology Conference 2018. 2018. p. 79.

[2] SELKOW, Stanley M. The tree-to-tree editing problem. Information processing letters, 1977, 6.6: 184-186.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 4, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Automata Approach to Approximate Tree Pattern Matching

Bc. Lukáš Renc

Department of Theoretical Computer Science
Supervisor: Ing. Eliška Šestáková

May 28, 2020

Acknowledgements

I would like to express my gratitude to my supervisor, Ing. Eliška Šestáková, for sharing her knowledge, words of encouragement, and her guidance through the program.

I would also like to thank my family: my parents and my brother. Last but not least I am incredibly thankful to my partner Terezka for unconditionally believing in me.

Declaration

Citation of this thesis

Renc, Lukáš. *Automata Approach to Approximate Tree Pattern Matching.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

This thesis is focused on approximate tree pattern matching via pushdown automaton. It uses tree edit operations defined by Selkow in [1]. The thesis studies methods of approximate tree pattern matching problem and propose of a new method for searching occurrences of tree pattern in an input tree with edit distance up to k and using pushdown automaton as a computational model. The method receives a tree pattern on the input and creates a pushdown automat for it. The automaton is then used for processing an input tree T— It searches occurrences of the tree pattern in the input tree. This thesis consists of theoretical background, the proposal of the method, implementation of the method, and its experimental testing. The method is implemented in Java programming language. The last part of the thesis are experiments that show the sensitivity of the method to the internal structure of both input trees.

Keywords finite automaton, pushdown automaton, approximate tree pattern matching, subtree matching, tree edit distance

Abstrakt

Tato práce se zabývá přibližným vyhledáváním ve stromech pomocí zásobníkového automatu. Využívá edit operace pro stromy, které byly definovány Selkowem v [1]. Práce v návaznosti na studium metod pro přibližné vyhledávání ve stromech představuje návrh nové metody pro vyhledávaní výskytů vzorového stromu uvnitř vstupního stromu s maximálně k chybami při použití zásobníkového automatu jako výpočetního modelu. Metoda nejdříve na vstupu dostane vzorový strom, jehož výskyt chceme vyhledávat. Dle něj sestaví zásobníkový automat. Poté pomocí nově vzniklého zásobníkového automatu zpracuje vstupní strom, uvnitř kterého se vyhledají výskyty vzorového stromu. Tato práce obsahuje popis stávajících metod a přístupů, návrh nové metody, implementaci nové metody a její otestování pro funkčnost a rychlost. Metoda je implementována v jazyce Java. Součástí práce jsou experimenty, na kterých jde vidět citlivost metody na vnitřní strukturu obou vstupních stromů.

Klíčová slova konečný automat, zásobníkový automat, přibližné vyhledávání, strom, editační vzdálenost

Contents

List of Figures

List of Tables

Introduction

Arbology is a computer science field that studies trees [2]. *Arbology* discipline applies the well-known principles of algorithms from *stringology* to trees to create effective analogous tree algorithms. The name *arbology* comes from a Spanish word *árbol* (in English: a tree). Every tree can be represented in a linear notation. In this thesis, we use a prefix bar notation, see [3]. Every instance of prefix bar notation can be generated by a context free grammar. Therefore, trees in prefix bar notation belong to context free languages. Hence a pushdown automaton is used as a model of computation. Because of the everyday need to compare, store, or search data, trees are used frequently. For example, trees are used for:

- Representing hierarchical data such as abstract syntax trees used by compilers [4].
- Storing data in a way that makes it efficiently searchable [5].
- Representing sorted lists of data [6].
- As a workflow for management systems [7].
- Storing Barnes-Hut trees used to simulate galaxies [8].

One of the fundamental tree problems is given two trees, show which changes need to be done to the first tree to match the second. A typical real-world example of the problem is: *"Find approximately this data inside this XML file"* This example task cannot be achieved by simple byte to byte comparison or hash value comparison. Text-based comparison methods (such as the UNIX utility `diff`) lack understanding of a rigid hierarchical structure in tree-like documents. Because of that, they produce unsatisfactory comparisons or irrelevant differences. Therefore advanced methods are needed. For example, Yang in [9] suggests possible solutions for program comparison. He makes use of the hierarchical structure of the programs. (Programming language specifies a rigid hierarchical structure to a program.) Programs can be seen as trees. In fact, they are trees—abstract syntax tree during the compilation process. Therefore, the problem of two programs comparison can be transformed into a tree to tree comparison problem. In most of the tree-like documents, each block of data has a specified position in the document. Hence in tree visualization, the order of vertices in a tree matter. Because of that, we use ordered trees.

An ordered tree is one of the higher-dimensional generalization forms of strings (others are, for example, graphs or webs). Therefore, a survey on string edit distance provides knowledge that can be adapted to tree edit distance. While string edit distance was studied five decades ago, tree edit distance was studied in the late eighties. Selkow [1] and Tai [10] describe tree edit distance as the minimum of specified edit operations (that apply changes to the first or the second input tree) to make the two trees match each other. Different sets of edit operations can be used.

Selkow in [1] works with labeled ordered trees and with three edit operations: insert a leaf, delete a leaf, relabel a vertex. There can be a specified cost for each operation. Each of the edit operations may be used recursively, so it is always possible to compute minimal edit distance. In literature, this edit distance is called 1-degree edit distance or just Selkow distance. He proposed a recursive algorithm to compute the edit distance between two trees. The time complexity of the algorithm is $\mathcal{O}(n \cdot m \cdot d)$, where n and m are the maximum numbers of children of any vertex in each of the trees, and d is the maximum depth of the trees.

Let P and T be ordered trees and k a maximal 1-degree edit distance. Šestáková, Melichar and Janoušek in [11] proposed a method for finding all occurrences of the tree pattern P in the input tree T with up to k edit operations between found occurrences and the tree pattern P. The method is based on a concept of a finite automaton. Its edit distance for the tree to tree comparison is based on the 1-degree edit distance. Firstly, it builds an automaton for a given tree pattern P. Secondly, tree T is used as an input to the automaton. Automaton outputs occurrences of the pattern P in the searched tree T. Edit distances between occurrences and the tree pattern P are up to k. However, this method does not support recursive deletion or insertion of leaves. Therefore, it is not always possible to calculate the edit distance between two trees via this method because of the finite automaton schema's limited computing power. The first phase of the method creates an automaton with $\mathcal{O}(|A|^k \cdot m^{k+1})$ states where A is used alphabet, $m = |T|$ and k is maximum given distance.

This thesis proposes a new method for approximate tree pattern matching based on automata theory and extends the paper [11]. The new method uses the Selkow approach presented in the paper *Tree to tree editing problem* [1].

Aim of the Thesis

The aims of this thesis are:

- Study methods for approximate tree pattern matching and the automata approach for constrained approximate subtree matching introduced in [11].
- Propose a new method for approximate tree pattern matching using the theory of formal languages and automata. For example, 1-degree edit distance and edit operations defined in [1].
- Implement the proposed method.
- Discuss time and space complexities.
- Perform appropriate testing of the implementation.

Structure of the Thesis

A brief description of the chapters and sections follows.

Chapter 1 (Theoretical Background) consists of used basic notation and definitions on alphabet, string, graphs, trees, finite and pushdown automata. In the end, there is an observation of the pushdown automata determinization problem.

Chapter 2 (Related Work) presents research done on related work to this thesis. It covers articles and papers on string pattern matching, approximate string pattern matching, tree to tree comparison problem, Selkow edit operations, and edit distance.

Chapter 3 (Using Pushdown Automata for Approx. Tree Pattern Matching) introduces a new approach to tree pattern matching and approximate tree pattern matching problem. The chapter firstly shows how this approach works with Selkow edit operations, how it is built into a pattern matching automaton, and how it is beneficial for approximate pattern matching. Secondly, the chapter shows a description of a new method that builds and uses a pushdown automaton for approximate pattern matching.

Chapter 4 (Implementation) presents the implementation of the proposed method. The implementation is meant as a *Poc* for the proposed method. The chapter consists of three sections. Firstly, the used space optimilisation is explained. Secondly, public methods of public classes are described. The third section describes implementation usage and the structure of an input file for the prepared run configuration.

Chapter 5 (Testing) The chapter is split into two sections. The first section tests correctness of the proposed method. It is focused on *Block type matching* and edit operation function tests. The second section examines the method performance for various input trees with different inner structures.

Chapter 6 (Conclusion) consists of a summary of this thesis, sums up fulfilled goals, and suggests enhancements for future work.

Chapter 1

Theoretical Background

In this chapter, all the notations through this thesis are given along with basic definitions and used terms.

1.1 Notations

Main notations used throughout this thesis are as follows:

- Σ for an alphabet,
- a, b, c for alphabet symbols,
- $\Sigma, b/c$ for a labeled transition in a pushdown automaton. Σ is an input symbol—every a from A, b is a stack top symbol to be popped from the stack, c is a symbol to be pushed to the stack,
- w, x, y, z for strings,
- L for a language,
- M for both a finite and a pushdown automaton,
- p, q for states,
- δ for an automaton transition function,
- q_0 for an initial state,
- F for a set of final states,
- R as a pushdown store alphabet,
- Z_0 as an initial pushdown store symbol,
- G for a graph,
- V for a set of vertices in a graph,
- E for a set of lists of edges in a graph,
- T for an input tree,
- P for a tree pattern,
- L_x for a level at position x of an automaton.

1.2 Basic Definitions

This section presents definition of terms that are used throghout this thesis. Section 1.2.1 defines terms in the same way as in [12]. Section 1.2.2 defines terms in the same way as in [13]. Section 1.2.3 defines terms in the same way as in [14] if not explicitly stated otherwise.

1.2.1 Alphabet, String, Language

Definition 1.1 (Alphabet). *An* alphabet *is a finite nonempty set of symbols.*

Definition 1.2 (Ranked alphabet). *A* ranked alphabet *is an alphabet where each symbol of a set has a unique nonnegative arity (or rank).*

Definition 1.3 (Arity of a symbol). *Given a ranked alphabet A, the* arity of a symbol $a \in A$ *is denoted $arity(a)$.*

Definition 1.4 (String). *A* string *over a given alphabet A is a finite sequence of symbols of A.*

Definition 1.5 (Length of a string). *A* length *of a string x, denoted by $|x|$, is the number of its symbols.*

Definition 1.6 (Empty string). *An* empty string *is an empty sequence of symbols denoted by ε.*

Definition 1.7 (Prefix). *A* prefix *of a string $x = x_1x_2, \ldots, x_n$ is a string $y = x_1x_2, \ldots, x_m$, where $m \leq n$.*

Definition 1.8 (Factor). *A* factor (substring) *of a string $x = x_1x_2, \ldots, x_n$ is a string $y = x_ix_{i+1}, \ldots, x_j$, where $1 \leq i \leq j \leq n$.*

Definition 1.9 (Subsequence). *A* subsequence *of a string $x = x_1x_2, \ldots, x_n$ is a string y obtained by deleting zero or more symbols from x.*

Definition 1.10 (Language). *A* language *L over an alphabet A is a set of strings over A.*

1.2.2 Graph

Definition 1.11 (Graph). *A* graph *G is a pair (V, E), where V is a set of vertices and E is a set of unordered pairs of vertices called edges. A pair $[m, n]$ indicates that there is an edge connecting vertices m and n.*

Definition 1.12 (Path). *A sequence of vertices $(n_0, n_1, \ldots, n_m)$, where $m \geq 1$, is a* path *of length m from the vertex n_0 to the vertex n_m if $\forall i \in \{1, 2, \ldots, m\}$ there is an edge connecting vertices n_{i-1} and n_i.*

Definition 1.13 (Cycle). *A* cycle *is a path $n_0, n_1, \ldots n_m$, where $n_0 = n_m$.*

Definition 1.14 (Connected graph). *A graph $G = (V, E)$ is* connected *when there is a path between every pair of vertices.*

Definition 1.15 (Directed graph). *A* directed graph *G is a pair (V, E), where V is a set of vertices and E is a set of ordered pairs of vertices called directed edges. A pair (m, n) indicates that for the vertex m, there is an edge leaving m and entering the vertex n.*

Definition 1.16 (Directed path). *A sequence of vertices $(n_0, n_1, \ldots n_m)$, where $m \geq 1$, is a* directed path *of length m from the vertex n_0 to the vertex n_m if $\forall i \in \{1, 2, \ldots, m\}$ there is a directed edge which leaves the vertex n_{i-1} and enters the vertex n_i.*

Definition 1.17 (Directed cycle). *A* directed cycle *is a directed path $n_0, n_1, \ldots n_m$, where $n_0 = n_m$.*

Definition 1.18 (Acyclic graph). *An* acyclic graph *is a graph that has no cycle.*

Definition 1.19 (Directed acyclic graph). *A* directed acyclic graph *is a directed graph that has no directed cycle.*

Definition 1.20 (Labeling). *A* labeling *of a graph $G = (V, E)$ is a mapping V into a set of labels.*

Definition 1.21 (Out-degree, in-degree). *Given a directed graph $G = (V, E)$ and a vertex $n \in V$, its* out-degree *is the number of distinct pairs $(n, m) \in E$, where $m \in V$. By analogy, the* in-degree *of vertex n is the number of distinct pairs $(m, n) \in E$ where $m \in V$.*

1.2.3 Tree

Definition 1.22 (Tree). *A* tree *is an acyclic connected graph.*

Definition 1.23 (Subtree). *A* Subtree *of a tree T is a tree consisting of a vertex in T and all of its descendants in T.*

Definition 1.24 (Rooted directed tree). *A* rooted and directed tree *T is a directed graph $T = (V, E)$ with a special vertex $r \in V$, called the* root*, such that (1) r has in-degree 0, (2) all other vertices of T have in-degree 1, (3) there is just one path from the root r to every vertex $n \in V$, where $n \neq r$.*

Definition 1.25 (Leaf). *Let $T = (V, E)$ be a rooted directed tree. Vertex $n \in V$ is called a* leaf *if it has out-degree 0.*

Definition 1.26 (Child). *Let $T = (V, E)$ be a rooted directed tree with vertices $m, n \in V$. A vertex n is a* child *of a vertex m if there is a directed edge $(m, n) \in E$.*

Definition 1.27 (Parent)**.** *Let $T = (V, E)$ be a rooted directed tree with vertices $m, n \in V$. A vertex m is a* parent *of a vertex n if there is a directed edge $(m, n) \in E$. Notation $parent(x)$ stands for "parent of vertex x". For example: $parent(n) = m$.*

Definition 1.28 (Sibling)**.** *Let $T = (V, E)$ be a rooted directed tree with vertices $m, n \in V$. A vertex n is a* sibling *of a vertex m if $parent(n) = parent(m)$.*

Definition 1.29 (Descendant)**.** *Let $T = (V, E)$ be a rooted directed tree with vertices $m, n \in V$. A vertex n is a* descendant *of a vertex m if there is a directed path $(m, \ldots, n)$.*

Definition 1.30 (Ancestor)**.** *Let $T = (V, E)$ be a rooted directed tree with vertices $m, n \in N$. A vertex m is an* ancestor *of a vertex n if there is a directed path $(m, \ldots, n)$.*

Definition 1.31 (Labeled tree)**.** *A* labeled *(rooted, directed) tree T is a (rooted, directed) tree where every vertex $n \in V$ is labeled by a symbol a of an alphabet A; the label of a vertex $n \in V$ is denoted $label(n)$. Labeling function is denoted as $label(T)$, where T is a labeled rooted tree. The label of a given tree T is the label of the root vertex of T. The labeling function returns a label of a given tree.*

Definition 1.32 (Ordered tree)**.** *An* ordered *(labeled, rooted, directed) tree is a (labeled, rooted, directed) tree where children $n_1, n_2, \ldots, n_m$ of a tree vertex n with an out-degree m are ordered.*

Definition 1.33 (Ranked tree)**.** *A* ranked *(rooted, directed) labeled tree is a (rooted, directed) labeled tree labeled by symbols from a ranked alphabet and out-degree of a vertex n labeled by symbol $a \in A$ is $arity(a)$.*

Definition 1.34 (Prefix bar notation [3])**.** *Prefix bar notation is a linear notation of trees. It is defined as follows:*

- *$prefbar(a) = a\,|$ if a is both the root and a leaf,*
- *$prefbar(T) = a\ prefbar(b_1)\ prefbar(b_2)\ \cdots\ prefbar(b_n)\ |$ if a is the root of the tree T and $b_1, b_2, \ldots, b_n$ are children of a.*

Example 1.1 (Prefix bar notation)**.** Let $T = (V, E)$ be an ordered labeled rooted tree shown in Figure 1.1.

The prefix bar notation of T is as follows:

$$prefbar(T) = a\,b\,c\,|\,d\,e\,|\,|\,|\,f\,g\,|\,h\,i\,|\,|\,|\,|$$

.

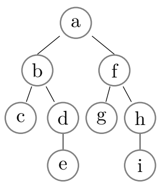


Figure 1.1: Rooted, ordered, labeled tree T.

Folowing labels for trees are used:

- *Input tree* is an ordered labeled tree in which a tree pattern is searched,
- *Tree pattern* is an ordered labeled tree whose oocurences are searched for in an input tree.
- Notation *tree* is used in the rest of this thesis as an ordered labeled rooted tree unless explicitly stated otherwise.

1.2.4 Finite and Pushdown Automaton

Definition 1.35 (Deterministic finite automaton)**.** *A deterministic finite automaton (DFA) is a quintuple $M = (Q, A, \delta, q_0, F)$, where*

- *Q is a finite set of states,*
- *A is an input alphabet,*
- *δ is a mapping from $Q \times A$ to Q,*
- *q_0 is the initial state,*
- *$F \subseteq Q$ is the set of final states.*

Definition 1.36 (Nondeterministic finite automaton)**.** *A nondeterministic finite automaton (NFA) is a quintuple $M = (Q, A, \delta, q_0, F)$, where*

- *Q is a finite set of states,*
- *A is an input alphabet,*
- *δ is a mapping from $Q \times A$ into the set of subsets Q (denoted by 2^Q),*
- *q_0 is the initial state,*
- *$F \subseteq Q$ is the set of final states.*

Definition 1.37 (Nondeterministic pushdown automaton)**.** *A nondeterministic pushdown automaton is a seven-tuple $M = (Q, A, R, \delta, q_0, Z_0, F)$, where*

- *Q is a finite set of states,*
- *A is an input alphabet,*
- *R is a pushdown store alphabet,*
- *δ is a mapping from $Q \times (A \cup \{\varepsilon\}) \times R$ into a set finite subsets of $Q \times R^*$,*

- *$q_0 \in Q$ is an initial state,*
- *$Z_0 \in R$ is the initial pushdown store symbol,*
- *$F \subseteq Q$ is the set of final (accepting) states.*

Definition 1.38 (Deterministic pushdown automaton)**.** *A pushdown automaton $M = (Q, A, R, \delta, q_0, Z_0, F)$ is deterministic, if the following holds*

- *$|\delta(q, a, \gamma)| \leq 1$, $\forall q, a, \gamma$ where $q \in Q, a \in (A \cup \{\varepsilon\}), \gamma \in R^*$,*
- *If $\delta(q, a, \alpha) \neq \emptyset, \delta(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$, then α is not a suffix of β and β is not a suffix of α (i.e., $\gamma\alpha \neq \beta$, $\alpha \neq \gamma\beta$),*
- *If $\delta(q, a, \alpha) \neq \emptyset, \delta(q, \varepsilon, \beta) \neq \emptyset$, then α is not a suffix of β and β is not a suffix of α (i.e., $\gamma\alpha \neq \beta$, $\alpha \neq \gamma\beta$).*

Pushdown automaton is a theoretical, computational model. Pushdown automata use the stack as computational memory with the following two operations: Push & Pop. Nondeterministic pushdown automata recognize context-free languages. There are some differences between finite and pushdown automata theories. For every non-deterministic finite automaton, there is an equivalent deterministic finite automaton. However, this does not apply to a family of pushdown automata. Some non-deterministic pushdown automata do not have their version of deterministic equivalent. Examples of such pushdown automata might be a pushdown automaton accepting palindromes written in the form like vv^R. The reason is that an automaton reads the palindrome from left to right, and the automaton cannot determine the center of the palindrome for it. A solution to a decision problem, if there is or not a deterministic version for a given nondeterministic pushdown automaton, is an open problem [15]. There are three classes of pushdown automata for which such a determinization is possible. These classes are called input-driven [16], visible [17] and height-deterministic pushdown automata [18].

Input driven pushdown automata were introduced in [19]. The idea of input driven pushdown automata is that the input symbols uniquely determine whether the automaton pushes a symbol, pops a symbol, or leaves the pushdown unchanged. The papers [20] and [21] studied further features of the class of input-driven pushdown automata languages. Deterministic and nondeterministic input driven pushdown automata are equally powerful. Input driven pushdown automaton is determinizable [22].

Visibly pushdown automaton is a pushdown automaton whose stack operations are determined by the input symbol it reads. Visibly pushdown automata are connected to nested words. A nested word is a linear structure (word) with a nesting relation formed by associating open-tags with their matching close-tags. A visibly pushdown automaton can reconstruct the nesting relation by pushing onto the stack on open-tags and popping from it close-tags [17]. Visibly pushdown automata, unlike pushdown automata, define a robust class of languages. The class is closed under all boolean operations, admits decidable procedures for problems such as inclusion and emptiness, and we can show that it is precisely as powerful as regular tree languages accepting the tree representation of the data. Visibly pushdown automata are determinizable [23].

Class of height deterministic pushdown automata consists of automata that, for any given input string, the stack height during any (nondeterministic) computation is a priori fixed. This class was introduced in [24]. Height deterministic pushdown automaton is determinizable [25].

Chapter 2

Related Work

This thesis proposes a new method for approximate tree pattern matching (in Chapter 3). The topics discussed in this chapter consists of fundamental ideas that are combined for the new method. This chapter consists of commented examples and illustrated explanations of concepts used in the newly proposed method. The chapter is split into three sections. They are as follows: (1) Tree Comparison and Related Problems, (2) String Pattern Matching, (3) 1-degree Edit distance. The first section, Section 2.1, discusses approaches to tree comparison, shows differences between tree and string comparison, and presents example usage of a hierarchical structure of trees for comparison purposes. Section 2.2 discusses a string pattern matching automaton model. It is described in detail. Also, a simulated run of the finite automaton on a sample string is presented. The pattern matching automaton model is used as a base in the proposed method. Section 2.3 presents Selkow's findings on tree-to--tree matching problem in great detail. His algorithm for tree comparison is presented with commented steps. Selkow's paper is a starting point for this thesis. His edit operations are used in the proposed method.

2.1 Tree Comparison and Related Problems

Tree to tree comparison problem is used in approximate tree pattern matching to determine edit distance between two trees. This section sums up findings from the field of tree theory and the usage of trees in practice. We focus on ordered labeled rooted trees as this is where this thesis aims to propose a new method. Because of the everyday need to compare or search data, the theory of trees and tree matching is an essential field in computer science. Yang in [9] further emphasizes the programmer's need to identify the differences between the two programs and suggests possible solutions. Two features can be exploited in a program to a program comparison. Firstly, a programming language specifies a rigid hierarchical structure to a program.

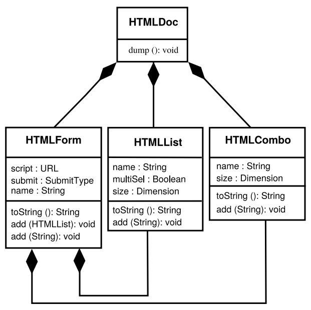


Figure 2.1: The base UML document—sample class diagram. Source: [26]

Secondly, programs can be seen as trees. (In fact, they are trees—abstract syntax tree during the compilation process.) Therefore, the problem of two programs comparison can be transformed into a tree to tree comparison problem. If tree theory is not used, the comparison produces unsatisfactory results or irrelevant differences. For example, text-based tools (such as the UNIX utility `diff`) lacks understanding for a rigid hierarchical structure in tree-like documents. Ohst in [26] sums up differences between string and tree matching problem. Algorithms for finding differences between particular kinds of structured documents like LaTeX, XML, HTML files, or data in CAD databases [27, 28] are often based on algorithms solving the tree-to-tree correction problem [29, 30, 10, 1]. These algorithms interpret documents as trees. Algorithms on trees try to find a sequence of edit operations that transform one tree into the other. Such sequences are called edit scripts (consisting of edit operations). The algorithms are based on different sets of edit operations. Every set includes basic operations for creating, deleting, or modifying a vertex of a tree. To further illustrate the problem, we continue by enclosing an example of finding differences between versions of a UML diagram.

UML diagram version The approach proposed by Ohst in [26] is based on the assumption that software documents are modeled in a fine-grained way, i.e., they are stored as syntax trees in XML files or a repository system. He proposed a computation algorithm that detects structural changes and enables their appropriate visualization. The example presented in [26] shows how to visualize changes to a document with a tree-like structure. The first three Figures 2.1,2.2, and 2.3 show gradual changes to an UML diagram. The base document is modified, see Figure 2.1. Firstly, by extending with inheritance, see Figure 2.2. Secondly, by adding an export feature, see Figure 2.3. The fourth Figure 2.4 shows all submitted changes together to the base document with respect to the hierarchical structure. The fifth Figure 2.5 shows the result (visualization of gradual changes to a UML document) of Ohst paper.

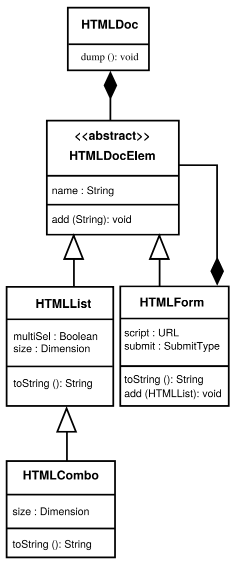


Figure 2.2: Class diagram of Figure 2.1 extended with inheritance. Source: [26]

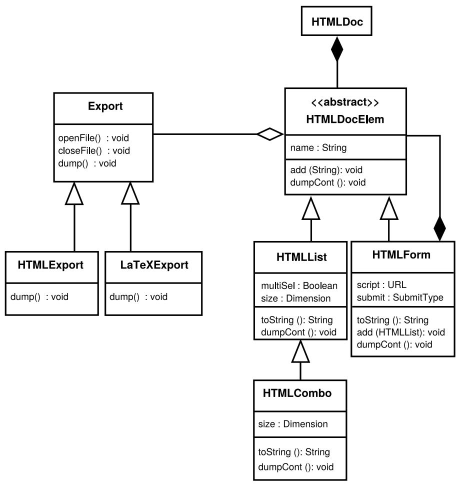


Figure 2.3: Class diagram from Figure 2.2 extended with export functionality. Source: [26]

Proposed solution in [26] results in the display shown in Figure 2.5. There are systematically marked changes displayed in one picture. Green color represents newly added features. On the contrary, in red are deleted links of the base document (effectively replaced by the green ones). In gray color is a standalone extension. Such a result cannot be achieved only by simple string comparison. A deeper understanding of

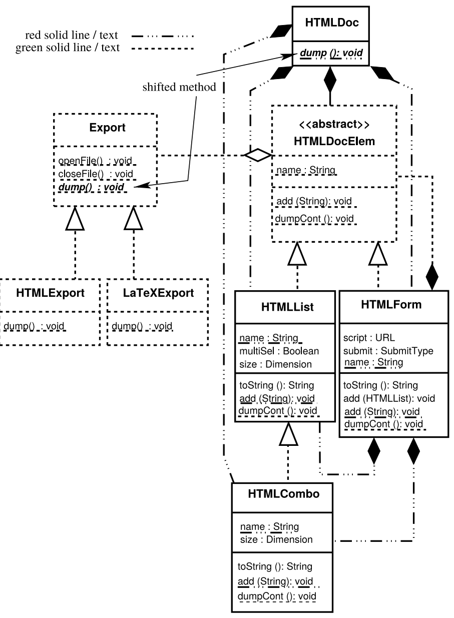


Figure 2.4: Unified document showing the differences between the base documents. Source: [26]

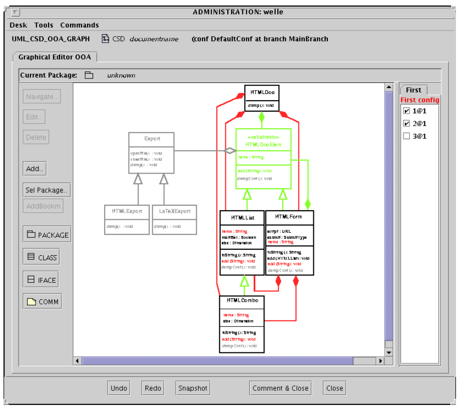


Figure 2.5: The result (visualisation of gradual changes to the base UML document) of the algorithm proposed in [26].

data structure and the exploitation of features of trees are needed to correctly calculate (and visualize) the difference.

Advanced Comparison Method Another example of tree usage used for the aim of this thesis is presented in [31]. Oommen came up with an advanced comparison method of the closeness of a target tree to other trees. The method is called *Noisy Sub-Sequence Tree Processing*. It compares the closeness of a target tree to other trees located in a database of trees. The method works in the following schema (as presented in [31]):

1. Calculate a constraint in respect of each tree in the database based on an estimated number of edit operations and a characteristic of the target tree.
2. Calculate a constrained tree edit distance between the target tree and each tree in the database using the constraint obtained in step (1)
3. Compare the calculated constrained tree edit distances.

Oommen proposed a method to compare the closeness of a tree pattern to other trees.

By definition, every subtree is itself a tree. A tree can be seen as a hierarchical data structure of subtrees. Therefore, Oommen's research can be adapted to a new method that searches for a given tree in all of the subtrees of another tree. Concept of the Oomen's algorithm is applied to the new tree pattern matching method proposed in this thesis.

2.2 String Pattern Matching

Pattern matching automaton belongs to a family of finite state automata (see Definition 1.2.4). In Example 2.6 there is a nondeterministic finite state automaton that is used for string pattern matching. It parses all strings whose suffix is "*bab*". Pattern matching automata are built offline, meaning the searched pattern P is known in advance. For each pattern P, there has to be a standalone pattern matching automaton. (These standalone automata can be joint later. [32, 12]) Pattern matching automata (especially determinized) can be memory intensive. However, searching for pattern P in text T via a pattern matching automaton runs in $\mathcal{O}(|T|)$. This is a typical space-time tradeoff in computer science. Space-time tradeoff is a case when algorithm trades increased space usage with decreased computational time.

The idea of pattern matching automaton is used for a new method that is proposed by this thesis.

Example 2.1 (Simulated Run of a pattern matching automaton for pattern $P = bab$)**.** Figure 2.6 shows a nondeterministic pattern matching automaton. For build construction details see [12, 33].

- Text $T =$ bbabbabab
- Pattern $P =$ bab

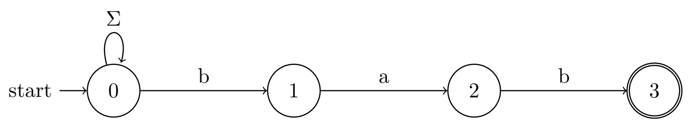


Figure 2.6: Pattern matching automaton A_1 for $P = bab$.

In this section, we simulate how pattern matching automaton A_1 that accepts all strings whose suffix is equal to $P = bab$ works with a text $T = bbabbabab$

Because of the (see Definition 1.1) Σ transition at the state 0, the text T is effectively split into these suffixes:

1. $T_{1,\dots,|T|}$ = bbabbabab
2. $T_{2,\dots,|T|}$ = babbabab
3. $T_{3,\dots,|T|}$ = abbabab
4. $T_{4,\dots,|T|}$ = bbabab
5. $T_{5,\dots,|T|}$ = babab
6. $T_{6,\dots,|T|}$ = abab
7. $T_{7,\dots,|T|}$ = bab
8. $T_{8,\dots,|T|}$ = ab
9. $T_{|T|}$ = b

These substrings are one by another, processed by A_1. Iterations of the automaton are counting. If any substring $T_{1,\dots,|T|}, \dots, T_{|T|}$ ends up in the state 3, the current count of iterations marks the position of an occurrence of pattern P in the string T.

Tables 2.1, 2.2, and 2.3 show the process of finding all occurrences of pattern P in the text $T = bbabbabab$. The first row of a table consists of input text T. The last row of a table consists of numbers which, if present, point to a position in text T where pattern P occurs. The remaining rows (the second, third, and fourth) in Table 2.4 consists of the states reached in the automaton by a substring of text T. Table 2.1 shows the first occurrence in text T =bbabbabab. Table 2.2 shows the second occurrence in text T =bbabbabab. Table 2.3 shows the third occurrence in text T =bbababbab. Table 2.4[1] sums up simulated work of the pattern matching automaton A_1.

[1]The table is a result of Basic Simulation Method, see [33]

substring(T)	•	b	a	b	b	a	b	a	b
Iteration counter	1	2	3	4	5	6	7	8	9

Table 2.1: Pattern matching automaton simulation—occurrence 1.

substring(T)	•	•	•	•	b	a	b	a	b
Iteration counter	1	2	3	4	5	6	7	8	9

Table 2.2: Pattern matching automaton simulation—occurrence 2.

substring(T)	•	•	•	•	•	•	b	a	b
Iteration counter	1	2	3	4	5	6	7	8	9

Table 2.3: Pattern matching automaton simulation—occurrence 3.

T	-	b	b	a	b	b	a	b	a	b
State	0	0	0	0	0	0	0	0	0	0
		1	1	2	1	1	2	1	2	1
					3			3		3
Position:					4			7		9

Table 2.4: Pattern matching automaton simulation—all occurrences.

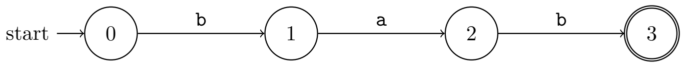


Figure 2.7: Pattern matching automaton for string $P_0 = bab$.

2.2.1 Approximate Pattern Matching Automata

Melichar in [34] proposes automata approach to approximate pattern matching. He makes use of the theory of finite automaton for approximate pattern matching. In this section, a variant of a finite automaton for approximate pattern matching is discussed. Hamming edit distance is used in Example 2.2. (The Hamming distance between two strings of equal length is the minimum number of substitutions required to change one string into the other. This edit distance has the only one edit operation—replace a symbol.)

Example 2.2. This example demonstrates automata usage for approximate pattern matching. Consider the sample string from the previous Example 2.1 $P = bab$.

- Firstly, we create a pattern matching automaton as in Example 2.1. As Hamming edit distance is specified for two strings with the same length, there is no loop transition at the first state. See Figure 2.7.

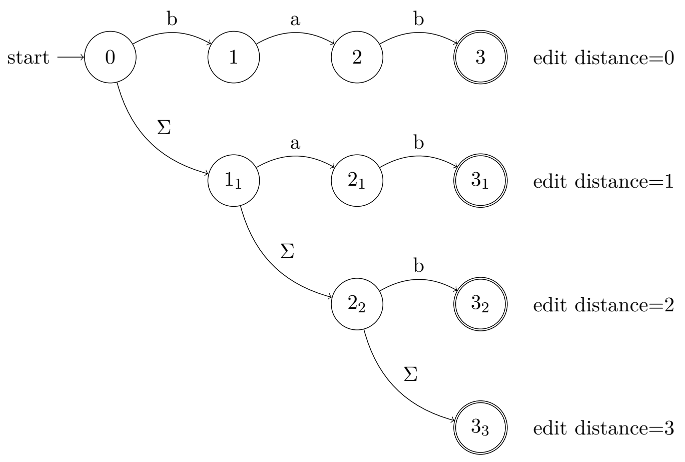


Figure 2.8: Approximate pattern matching automaton for $P = bab$.

- Secondly, we add a supported edit operation into the automaton. This is done by connecting it to a pattern matching automatons for strings P_1 = ab and P_2 = a. These are connected via Σ transitions (Definition 1.1), see Figure 2.8.

  Usage of additional levels works as a simple incremental counter. Every level represents a value of the total edit distance. By every step to a lower level in an automaton, edit distance is increased by one. (See [34] for formal definition.)

  Decision problem version of the automaton, with the specified maximal edit distance equals to k (decide if edit distance between P and T is smaller than k), can be easily created by allowing only k levels in the automaton and checking if the automaton finishes in accepting state.

2.3 1-degree Edit Distance and Related Problems

This section sums up Selkow's paper *Tree to tree editing problem* [1] in which 1-degree edit distance was introduced. Selkow proposes a method for the tree to tree editing problem. The paper proposes a solution that compares two trees and shows which changes need to be done, so the trees match. He works with rooted labeled ordered trees. (For labeled ordered tree definition see Definition 1.31 and 1.32.) Labeling function is denoted as $label(X)$, where X is a labeled rooted ordered tree. The labeling function returns a label of a given tree. Selkow defines three different operations. They are: (1) Insert a leaf (2) Delete a leaf (3) Relabel a vertex. Each of the edit operations may be used recursively. Therefore, it is always possible to compute minimal 1-degree edit distance. Only those three edit operations may be used to transform trees to match each other. By applying this restriction to edit operations set, the proposed method yields into an algorithm that is transparent and runs in polynomial time. The algorithm in pseudocode is presented in Section 2.3.1. Here, we present definitions used in Selkow's approach, show how Selkow manipulates via the edit operations with trees, and discuss three fundamental theorems used by the Selkow method.

Formally edit operations are defined as follow:

Definition 2.1 (Relabel Operation). *Given a tree with label*$(T) = s_j$ *and subtrees* $T_1, \ldots, T_m$*: A label change operation Relabel*(s_j, s_k) *applied to* T *yields the tree* T^* *with label*$(T^*) = s_k$ *and subtrees* $T_1, \ldots, T_m$.

In other words, the structure of the relabeled tree T remains the same, only the label of the root vertex of the tree T changes from s_j to s_k.

Definition 2.2 (Insert Operation). *Given a tree with label*$(T) = s_j$ *and subtrees* $T_1, \ldots, T_m$*:*

For $0 \geq i \geq m$ *and tree* A*, an insert operation* $Insert(A)$ *applied to* T *at* i *yields the tree* T^* *with label*$(T^*) = s_j$ *and subtrees* $T_1, \ldots, T_i, A, T_{i+1}, \ldots, T_m$.

In other words, the structure of the tree T is extended by adding a subtree A into i^{th} position between the subtrees of the tree T.As this operation is only defined for leaves, it is always possible to perform such action without affecting other vertices of the tree T.

Definition 2.3 (Delete Operation). *Given a tree with label*$(T) = s_j$ *and subtrees* $T_1, \ldots, T_m$*:*

For $1 \geq i \geq m$ *, a delete operation Delete*(T_i) *applied to* T *at* i *yields the tree* T^* *with label*$(T^*) = s_j$ *and subtrees* $T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_m$.

In other words, the structure of the tree T is reduced by taking out the i^{th} subtree. As this operation is only defined for leaves, it is always possible to perform such action without affecting other vertices of the tree T.

Definition 2.4 (Edit Operation). *An edit operation is any of the above three operations.*

Definition 2.5 (Cost of edit operation). *We associate a nonnegative cost with each edit operation in the following manner. Associated with each pair of labels* (s_i, s_j) *is a cost* $C_{Relabel}(s_i, s_j)$ *of applying the operation Relabel*(s_i, s_i)*. For each label* s_i*, we let* $C_{Insert}(s_i)$ *and* $C_{Delete}(s_i)$ *denote the costs of applying operations Insert(T) and Delete(T) respectively, where T is a tree with one vertex with label* s_i*. For an arbitrary tree T, let* $c_{Insert}(T) = \sum_{v \epsilon T} c_{Insert}(labelOf(v))$ *and* $c_{Delete}(T) = \sum_{v \epsilon T} c_{Delete}(labelOf(v))$*. For any three labels* s_i, s_j *and* s_k *we assume* $C_{Relabel}(s_i, s_i) = 0$ *and* $C_{Relabel}(s_i, s_j) \geq C_{Relabel}(s_i, s_k) + C_{Relabel}(s_k, s_j)$.

In other words, every edit operation has it's specified cost. Cost is a non--negative numeric value. The cost of relabeling any vertex to its own label is set to 0. It is not possible to decrease the total cost of edit operations by repetitive relabeling vertices.

Definition 2.6 (Edit Tree distance)**.** *Given any trees A and B and the set of sequences of edit operations which when applied to A yield a tree equal to B, we let $\sigma(A, B)$ denote the minimum of the sums of the costs of each sequence. If tree A has subtrees $A_1, \ldots, A_m$ and tree B has subtrees $B_1, \ldots, b_n$, then $\sigma(A, B) \leq C_{Relabel}(label(A), label(B)) + \sum_{i=1}^{m} C_{Delete}(A_i) + \sum_{i=1}^{n} C_{Delete}(B_i)$.*

Selkow also shows the validity of theorems[2] 2.1, 2.2, and 2.3. The algorithm is based on these theorems. The first two theorems show how to calculate total cost when comparing a vertex to an arbitrary tree. The third theorem says about comparing two arbitrary trees: A and B. The total cost value is at maximum equal to relabeling the roots plus deleting all the vertices from the first and inserting all the vertices from the second one.

Theorem 2.1. *For any tree A with subtrees $A_1, \ldots, A_m$ $(m \geq 0)$ and tree B with subtrees $B_1, \ldots, B_n$ $(n \geq 0)$: $\sigma(A(0), B(j)) = C_{Relabel}(label(A), label(B)) + \sum_{k=1}^{i} C_{Insert}(B_k)$.*

Theorem 2.2. *For any tree A with subtrees $A_1, \ldots, A_m$ $(m \geq 0)$ and tree B with subtrees $B_1, \ldots, B_n$ $(n \geq 0)$: $\sigma(A(i), B(0)) = C_{Relabel}(label(A), label(B)) + \sum_{k=1}^{i} C_{Delete}(A_k)$*
for $0 \leq j \leq n$ and $0 \leq i \leq m$.

Theorem 2.3. *For any tree A with subtrees $A_1, \ldots, A_m$ $(m \geq 0)$ and tree B with subtrees $B_1, \ldots, B_n$ $(n \geq 0)$:*

$$\sigma(A(0), B(j)) = min \begin{cases} \sigma(A(i-1), B(j-1)) + \sigma(A(i), B(j)) \\ \sigma(A(i), B(j-1)) + C_{Insert}(B(j)) \\ \sigma(A(i-1), B(j-1)) + C_{Delete}(A(i))abc \end{cases}$$

for $1 \leq i \leq m$ and $1 \leq j \leq n$.

For better understanding, we present examples of 1-degree edit distance. In all examples tree A illustrated in Figure 2.9 is considered. The examples consist of pairs of trees. On the left side of each pair, there is a sample tree B. An edit operation is applied to the tree B to match the tree A. On the right side of each pair, there is the tree B after applying edit operation/operations. Edited tree B (right side of a pair) matches the tree A. Each edit operation is densely dotted and highlighted in a different color. Inserted vertices are highlighted in the green color. Deleted vertices are highlighted in the red color. Relabeled vertices are highlighted in the blue color.

[2]For proofs, see [1].

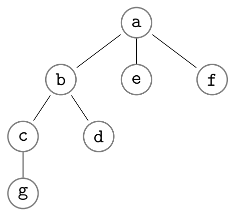


Figure 2.9: Sample tree A.

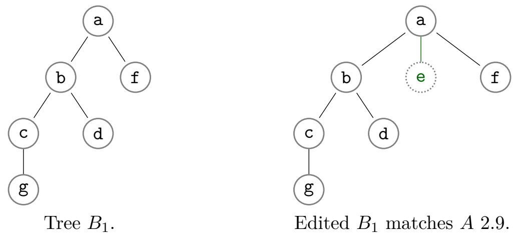


Figure 2.10: 1-degree edit distance = 1; Insert.

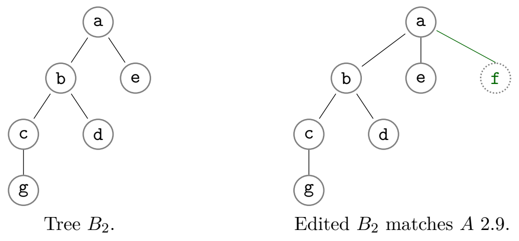


Figure 2.11: 1-degree edit distance = 1; Insert.

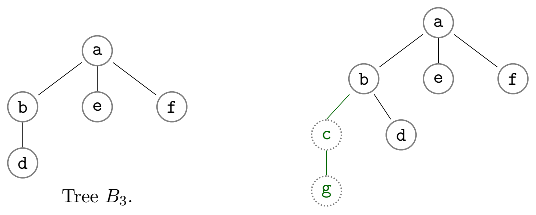
Example 2.3. Insert edit operation is presented in multiple figures in this example. See Figures 2.10, 2.11, and 2.12.

Example 2.4. Delete edit operation is presented in multiple figures in this example. See Figures 2.13, 2.14, and 2.15.

Example 2.5. Relabel edit operation is presented in multiple figures in this example. See Figures 2.16, and 2.17.

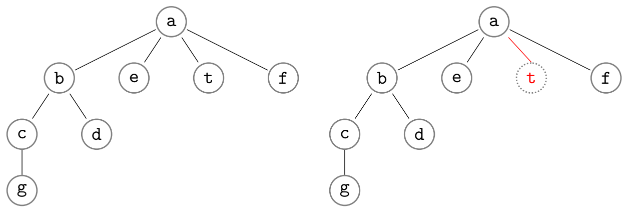
2.3.1 Tree to Tree Editing Problem Algorithm

Selkow's algorithm computes the minimal edit distance between two trees. It allows only leaf insertion, deletion, and relabel a vertex operation. By applying these restrictions, the algorithm is straightforward yet effective (the algorithm is polynomial in respect to the height of the input trees). The algorithm is recursive; it outputs tree edit distance. Trees to be compared are on the input. It starts with both root vertices of the two compared trees.



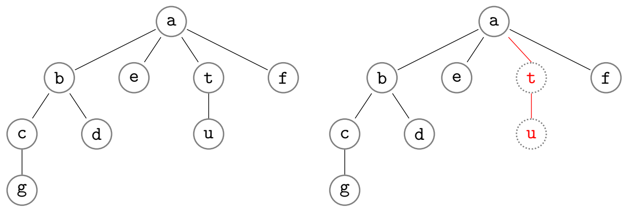
Tree B_3. Edited B_3 matches A 2.9.

Figure 2.12: 1-degree edit distance = 2; Insert.



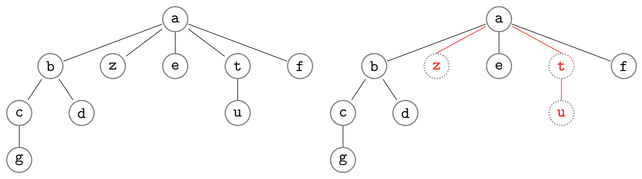
Tree B_4. Edited B_4 matches A 2.9.

Figure 2.13: 1-degree edit distance = 1; Delete.



Tree B_5. Edited B_5 matches A 2.9.

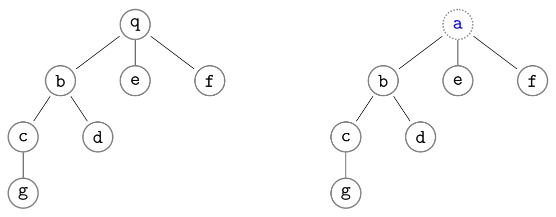
Figure 2.14: 1-degree edit distance = 2; Delete.



Tree B_6. Edited B_6 matches A 2.9.

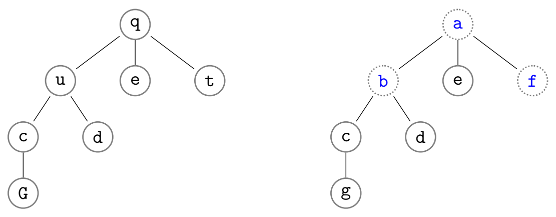
Figure 2.15: 1-degree edit distance = 3; Delete.

The algorithm is then called with each child of the first root vertex and each child of the other root. In each recursive step, it uses a temporary matrix in which it stores intermediate results— minimum cost to edit the subtrees. The time complexity of the algorithm is $\mathcal{O}(n \cdot m \cdot d)$, where n, m are the maximum degree of the first tree, respectively the second and d is the maximum depth of the trees.



Tree B_7. Edited B_7 matches A 2.9.

Figure 2.16: 1-degree edit distance = 1; Relabel.



Tree B_8. Edited B_8 matches A 2.9.

Figure 2.17: 1-degree edit distance = 3; Relabel.

Notation used in Example 2.6

- Let X be a vertex in an arbitrary tree A. Let Y be a vertex in arbitrary tree B. Notation used in this example reflects this fact as X_A and Y_B.
- Edit operation $insert_A(Y_B)$ returns the cost of inserting the vertex Y_B from the tree B into the tree A. Insert edit operation always inserts a vertex from the second tree into the first.
- Edit operation $delete_A(X_A)$ returns the cost of deleting the vertex X_A from the tree A. Delete edit operation always deletes a vertex from the first tree.
- Edit operation $relabel(X_A, Y_B)$ returns the cost of relabeling the vertex X_A from the tree A to label of the vertex Y_B from the tree B.
- Notation $slkwDst(X_A, Y_B)$ is a recursive call of the algorithm itself with the trees (X_A, Y_B) as an input.

Example 2.6 (Simulated run of the algorithm)**.** As input for Algorithm 1 a tree pattern P and an input tree T are used. Both of them are shown in Figure 2.18. Tree T is edited to match the tree pattern P. (Edited tree T is also shown in Figure 2.18.)

Algorithm 1: 1-degree Edit Distance algorithm.

```
Data: Tree $A$, Tree $B$
Result: Edit distance between tree $A$ and $B$
$m = Degree(A)$;
$n = Degree(B)$;
$Matrix\ D[][] = new\ [0, \dots, m][0, \dots, n]$;
$D[0][0] = c_{Relabel}(label(A), label((B))$;
for $i = 1;\ i \leq m;\ i++$ do
    $D[i][0] = D[i-1][0] + c_{Delete}(A_i)$;
end
for $j = 1;\ j \leq n;\ j++$ do
    $D[0][j] = D[0][j-1] + c_{Insert}(B_j)$;
end
for $i = 1;\ i \leq m;\ i++$ do
    for $j = 1;\ j \leq n;\ j++$ do
        $D[i][j] = min(\ D[i-1][j-1] + slkwDst(A_i, B_i),\ D[i][j-1] + c_{Insert}(B_j),\ D[i-1][j] + c_{Delete}(A_i))$;
    end
end
return $D[m][n]$;
```

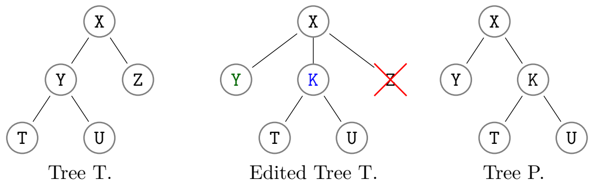


Figure 2.18: Sample trees used for simulated run of the algorithm.

1. Create a matrix $D[m][n]$. The matrix is initilized with following values, see Table 2.5.

- $D[0][0] = relabel(root_T, root_P) = 0$
- $D[0][1] = delete_T(T_T, U_T, Y_T) = 3$
- $D[0][2] = delete_T(\{T_T, U_T, Y_T\} + Z_T) = 4$
- $D[1][0] = insert_T(Y_P) = 1$
- $D[2][0] = insert_T(Y_P + \{T_P, U_P, K_P\}) = 4$

0	3	4
1		
4		

Table 2.5: Matrix D in initiall state.

0	3	4
1	2	3
4	2	3

Table 2.6: Matrix D fully computed.

2. Compute matrix D.

   a) To make the example more transparent, we substitute reccurent calls with precomputed results.

      - $slkwDst(Y_T, Y_P) = delete_T(T_T) + delete(U_T) = 2$
      - $slkwDst(Y_T, K_P) = relabel(Y_T, K) = 1$
      - $slkwDst(Z_T, Y_P) = relabel(Y_T, K) = 1$
      - $slkwDst(Z_T, K_P) = relabel(Y_T, K) + ins(T) + ins(U) = 3$

   b) Fill matrix D with precomputed results, see Table 2.6.

      - $D[1][1] = min \left\{ \begin{array}{l} D[0][0] + slkwDst(Y_T, Y_P) = 0 + 2 \\ D[0][1] + insert_T(Y_P) = 3 + 1 \\ D[1][0] + delete_T(T_T, U_T, Y_T) = 1 + 3 \end{array} \right\}$
      - $D[2][1] = min \left\{ \begin{array}{l} D[1][0] + slkwDst(Z_T, Y_P) = 1 + 1 \\ D[1][1] + insert_T(T_P, U_P, K_P) = 2 + 3 \\ D[2][0] + delete_T(T_T, U_T, Y_T) = 4 + 3 \end{array} \right\}$
      - $D[1][2] = min \left\{ \begin{array}{l} D[0][1] + slkwDst(Y_T, K_P) = 3 + 1 \\ D[0][2] + insert_T(Y_P) = 4 + 1 \\ D[1][1] + delete_T(Z_T) = 2 + 1 \end{array} \right\}$
      - $D[2][2] = min \left\{ \begin{array}{l} D[1][1] + slkwDst(Z_T, K_P) = 2 + 3 \\ D[1][2] + insert_T(T_P, U_P, K_P) = 3 + 3 \\ D[2][1] + delete_T(Z_T) = 2 + 1 \end{array} \right\}$

0	→	3	→	4
↓	↘			
1		2	→	3
↓	↘			
4		2	→	3

Table 2.7: Solution visualized in matrix D.

3. Return result.

   Found sequence of edit operations is reflected in matrix D, see Table 2.7.

   $$insert_T(Y_P), relabel(Y_T, K_P), delete_T(Z_T)$$

   Return $D[3][3] = 3$.

Chapter 3

Using Pushdown Automata for Approximate Tree Pattern Matching

This chapter introduces a novel automata-based method for searching for approximate matches of a tree pattern in an input tree. The problem of edit distance between two trees is a subproblem to approximate tree pattern matching problem. The proposed method uses Selkow edit operations, see Section 2.3.

The first section (Section 3.1) in this chapter introduces and defines a new approach for claiming a found pattern occurrence in the field of tree matching problem. In the second section (Section 3.2), the approach from the first section is further extended to the field of approximate tree searching problem. The last section (Section 3.3) proposes a new method based on the approach from Sections 3.2, 3.3. The proposed method creates a pushdown automaton that is used for approximate tree pattern matching.

Similarly as in string matching we use the following notations:

- P for a tree pattern.
- T for an input tree.

3.1 Block Type Matching

There are several approaches for tree pattern matching that search an input tree up to its leaves. However, these approaches do not solve problems presented in Examples 3.1, and 3.2. Therefore, we propose a new approach to tree pattern matching to address such problems. We call it *Tree block type pattern matching* that is throughout the rest of this thesis abbreviated to *Block type matching.*

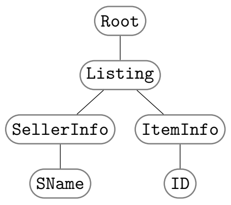


Figure 3.1: Tree pattern P.

Definition 3.1 (Block edit operation)**.** *Let $T = (V, E)$ be a labeled unranked rooted ordered tree with label$(T) = s_j$ and subtrees $T_1, \ldots, T_m$:*

For $1 \geq i \geq m$, if T_i is a leaf and is either the left-most or the right-most child of any vertex T_i then the block edit operation BlckEditOp(T_i) applied to T at i yields the tree T^* with label$(T^*) = s_j$ and subtrees $T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_m$.

In other words, Block edit operation works similarly to Delete operation, see Definition 2.3. However, a vertex can be deleted via Block edit operation only if the vertex is a leaf and is the left-most, respectively the right-most descendant of its parent.

Definition 3.2 (Tree block type pattern matching)**.** *A tree pattern $P = (V_P, E_P)$ matches an input tree $T = (V_T, E_T)$ in a vertex $n \in N_T$ if the subtree of T rooted at n modified by block edit operations matches P.*

The block edit operation deletes a vertex in the same way as Delete operation, but the cost of the Block edit operation is zero. Therefore, the Block edit operation does not require an occurrence of a searched pattern in an arbitrary tree T to be a subtree of T.

When we search for an occurrence with a minimal edit distance k, we prefer using the Block edit operation rather than the Delete operation because the cost of Block edit operation is lower than the Delete edit operation cost. Therefore, the set of vertices with Block edit operation applied to them is disjoint with the set of vertices with Delete operation applied to them. These relaxations are useful for everyday search requirements. Typically, when a user searches for particular information and does not know the whole context of it.

Example 3.1. An example use-case of *Block type matching* is searching in an XML document. The structure shown in Figure 3.2 of an XML document is from a real-world data from eBay auctions.

Let suppose a user is only interested in all vertices labeled as `ID` and `SName`. This query can be represented as a tree, see Figure 3.1. The information the user is searching for is the tree pattern P. The input tree T is the structure of the whole document presented in Figure 3.2. The goal is to find all occurrences of P in T. There are many other categories in the input tree T

```
Root
├── Listing ........................ First listing-root vertex of an offer
│   ├── Seller Info ........................ Infomartion about a seller
│   │   ├── SName ........................................ Seller's name
│   │   │   ├── GivenN ..................................... Given name
│   │   │   └── SurN ........................................ Surname
│   │   └── SRating ..................................... Seller's rating
│   ├── Item Info .................. Information obout an offered item
│   │   ├── ID ................................... Identication number
│   │   └── Battery ...................... Description of offerred battery
│   │       └── Capacity ..................... Capacity of offered item
│   └── Payment .......................... Supported type of payment
├── Listing ...................... Second listing-root vertex of an offer
│   ├── Seller Info ........................ Infomartion about a seller
│   │   ├── SName ........................................ Seller's name
│   │   │   ├── GivenN ..................................... Given name
│   │   │   └── SurN ........................................ Surname
│   │   └── SRating Info ................................ Seller's rating
│   ├── Item Info .................. Information obout an offered item
│   │   ├── CPU ................................... Type of offered CPU
│   │   │   └── Vendor ............................. Vendor of the CPU
│   │   └── ID ................................... Identication number
│   └── Payment .......................... Supported type of payment
└── Listing ....................... Third listing-root vertex of an offer
    ├── Payment .......................... Supported type of payment
    ├── Seller Info ........................ Infomartion about a seller
    │   ├── SRating Info ................................ Seller's rating
    │   └── SName ........................................ Seller's name
    │       ├── GivenN ..................................... Given name
    │       └── SurN ........................................ Surname
    └── Item Info .................. Information obout an offered item
        ├── ID ................................... Identication number
        └── Color ................................ Color of offered item
```

Figure 3.2: Sample XML structure.

that the user is not interested in (labels were truncated). For example, item info contains different subtrees for each product. The result of the example for the occurrences of the tree pattern P in the input tree T is shown in Figure 3.3. The searched information is highlighted in blue. The vertices deleted by *Block type operation* are crossed out.

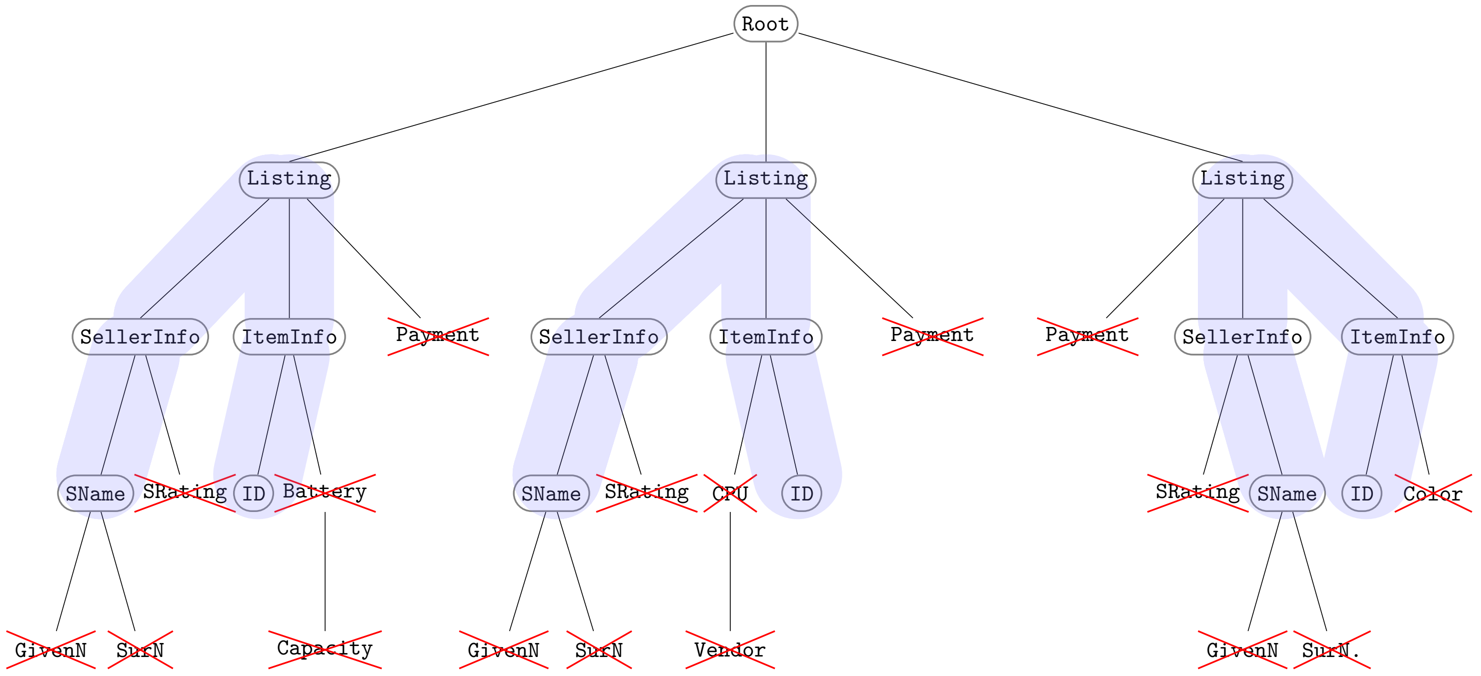


Figure 3.3: Real world auction data. Input tree T with highlighted occurrences of tree pattern P from Figure 3.1.

Example 3.2. Another example use-case of *Block type matching* is searching in an HTML document for specific tags. If the users wants to find all paragraphs `<p>` which contain two sections in bold `<b>`. We do not want to specify:

- which specific elements are in bold `<b>` sections,
- which element is the paragraph `<p>` enclosed in,
- if paragraph `<p>` has any other descendant element next to the the wanted two bold `<b>` sections.

Theorem 3.1. *Let $P = (V_P, E_P)$ and $T = (V_T, E_T)$ be two labeled unranked rooted ordered trees, respectively. Let $O = \{(V_{O1}, E_{O1}), \cdots, (V_{On}, E_{On})\}$ be a set of all occurences of P in T. Let $V_R = V_T \setminus \{V_{O1}, \cdots, V_{On}\}$ be a set of vertices. Block type matching allows any occurence $O_i \in O$ to have a vertex $v \in V_R$ as a descendant of any vertex $u \in V_{Oi}$.*

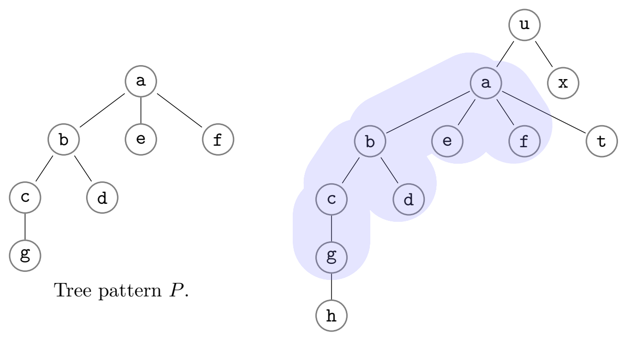
In other words, an occurrence of P in T found via *Block type matching* can have additional subtrees below itself.

Proof. Let $P = (V_P, E_P)$ and $T = (V_T, E_T)$ be two isomorfic labeled unranked rooted ordered trees, respectively. Let X be an arbitrary labeled unranked rooted ordered tree. Let T_2 be a tree that is created by adding X as a descendant to any leaf $a \in N_T$. Theorem 3.1 says that there is a *Block type matching* occurrence of P in T_2. Definition 3.2 allows repetetive deleting of either the left-most, or the right-most leaf in the input tree. Therefore, it is always possible to recursively delete all vertices of X in T_2. $T_2 \setminus X$ is isomorfic to T, T is isomorfic to P. Occurrence of P in T is claimed. $\square$

Theorem 3.2. *Let $P = (V_P, E_P)$ be a labeled unranked rooted ordered tree. Let $T = (V_T, E_T)$, where $V_T = V_P$ and $E_T = E_P$ be a labeled unranked rooted ordered tree. For each vertex $v \in V_T : u_1 \dots u_n \in V_T, \{v, u_1\}, \dots, \{v, u_n\} \in E_P$ (u_1 ...u_n are ordered children of v) add vertices u_0 and u_{n+1} : $\{v, u_0\}, \{v, u_1\}, \dots, \{v, u_n\}, \{v, u_{n+1}\} \in E_T$ of the tree T as the new left-most, respectively right-most children of vertex v. Block type matching finds an occurence of P in the modified T.*

In other words, *Block type matching* occurrence of P in T can have an additional sibling on the left/right side.

Proof. Let $P = (V_P, E_P)$ and $T = (V_T, E_T)$ be two isomorfic labeled unranked rooted ordered trees, respectively. Let X be an arbitrary labeled unranked rooted ordered tree. Let T_2 be a tree that is created by adding X as the left-most descendant to any vertex $a \in N_T$, the right-most descendant respectively. Theorem 3.2 says that there is a *Block type matching* occurrence of P in T_2. Definition 3.2 allows repetetive deleting of either the left-most, or the



Tree pattern P.

Found exact occurrence of P in T.

Figure 3.4: Example of Block Type Matching.

right-most descendant in the input tree. Therefore, it is always possible to recursively delete all the vertices of X in T_2. $T_2 \setminus X$ is isomorfic to T, T is isomorfic to P. Occurrence of P in T is claimed. □

Example 3.3. The example shows a concrete case of applying *Block type matching* to a tree-search problem. Figure 3.4 consists of a pair of trees. On the left side, there is a sample tree pattern P we search for. On the right side, there is an input tree T in which we search. The *Block type matching* is applied. Exact matching is used. The occurrence of tree pattern P found in the input tree T is highlighted in a light blue color in Figure 3.4.

3.2 Approximate Block Type Matching

The goal of this thesis is to propose a method for approximate tree pattern matching. We combine *Block type matching* and Selkow edit operations to achieve the goal. This combination is called Approximate block type matching. Concept and implementation of *Block type matching* provide a possibility to use Selkow edit operation Delete instead of it. Therefore the proposed method in this thesis is functional and useful even without *Block type matching.* Approximate block type matching uses the Selkow's edit operations (as presented in 2.3). Because *Block type matching* only specifies conditions under which an occurrence is claimed to be found, the effects of the Selkow's operations remain the same.

Definition 3.3 (Block edit distance)**.** *Let $P = (V_P, E_P)$ and $T = (V_T, E_T)$ be two labeled unranked rooted ordered trees. Block edit distance is a function $BlckDist(P, T)$ that returns minimal cost of edit operations in total to transform T to P. Tree T is modified by Selkow edit operations, see Definition 2.3, and by the block edit operation, see Definition 3.1. Block edit operation cost value equals to zero.*

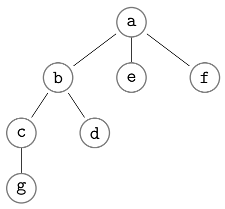


Figure 3.5: Tree pattern P.

Definition 3.4 (Approximate tree block type pattern matching with maximum of k errors)**.** *A tree pattern $P = (V_P, E_P)$ matches an input tree $T = (V_T, E_T)$ in a vertex $n \in N_T$ if the block edit distance (BlckDist) between the pattern P and the subtree of T rooted at n is less than or equal to k, block type matching is applied. i.e., $BlckDist(P, T_n) \leq k$.*

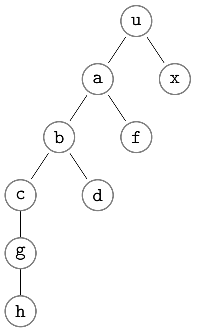
Examples 3.4, 3.5, and 3.6 show Approximate tree block type pattern matching with k errors. In other words these examples show combinations of *Block type matching* and Selkow edit operations. In all of the examples below, the tree pattern P from Figure 3.5 is searched in various input trees.

The examples are divided into three categories by an edit operation. Multiple examples of a specific edit operation and *Block type matching* is presented in each of them. Each edit operation is highlighted in a different color. Inserted vertices are highlighted in the green color. Deleted vertices are highlighted in the red color. Relabeled vertices are highlighted in the blue color. The occurrence of the tree pattern P found in an edited input tree T_i (i is an index) is highlighted in a light blue color.

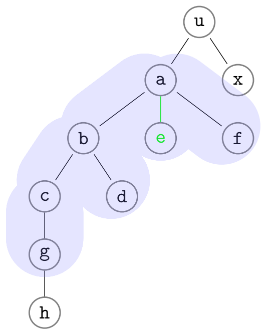
Example 3.4 (Insert edit operation combined with *Block type matching*)**.** The edit operation is presented here in multiple examples. See Figures 3.6, 3.7, and 3.8.

Example 3.5 (Delete edit operation combined with *Block type matching*)**.** The edit operation is presented here in multiple examples. Figures 3.9, and 3.10 show difference in input tree that results in using Delete edit operation instead of *Block type matching.* See Figures 3.9, 3.10, 3.11, and 3.12.

Example 3.6 (Relabel edit operation combined with *Block type matching*)**.** The edit operation is presented here in multiple examples. See Figures 3.13, and 3.14.



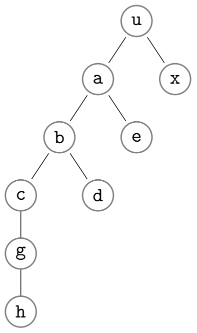
Tree T_1.



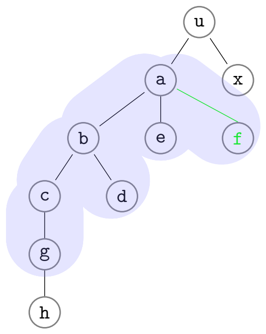
Edited T_1 block matches P from Figure 3.5.

$$BlckDist(P, T_1) = 1;\ \ Editscript = Ins(e).$$

Figure 3.6: Pattern P is in T_1 at vertex `a`, edit distance = 1.



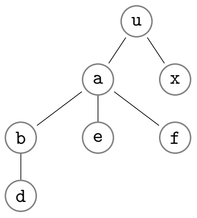
Tree T_2.



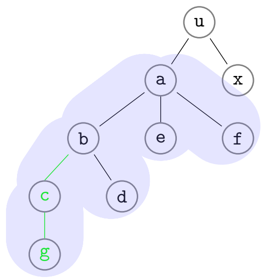
Edited T_2 block matches P from Figure 3.5.

$$BlckDist(P, T_2) = 1;\ \ Editscript = Ins(f).$$

Figure 3.7: Pattern P is in T_2 at vertex `a`, edit distance = 1.



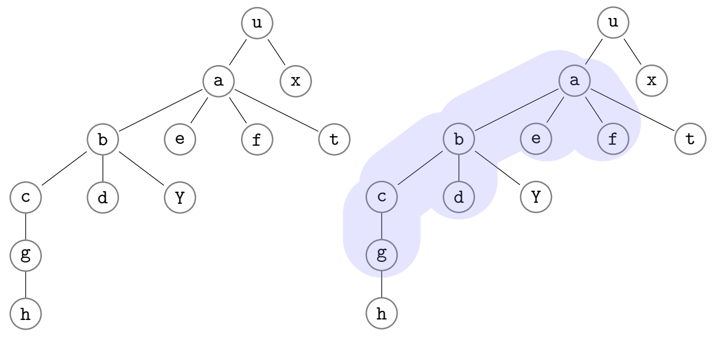
Tree T_3.



Edited T_3 block matches P from Figure 3.5.

$$BlckDist(P, T_3) = 2;\ \ Editscript = Ins(c), Ins(g).$$

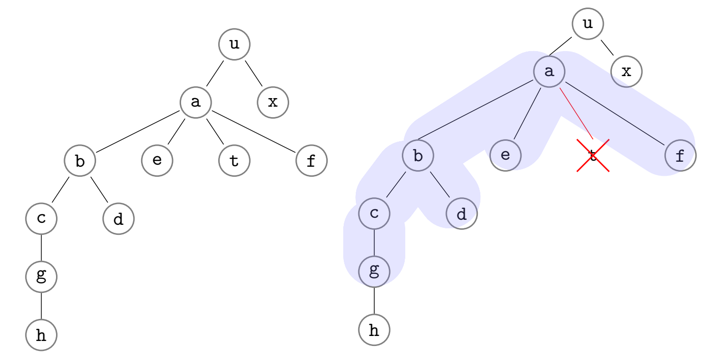
Figure 3.8: Pattern P is in T_3 at vertex `a`, edit distance = 2.



Tree T_4. T_4 block matches P from Figure 3.5.

$$BlckDist(P, T_4) = 0.$$

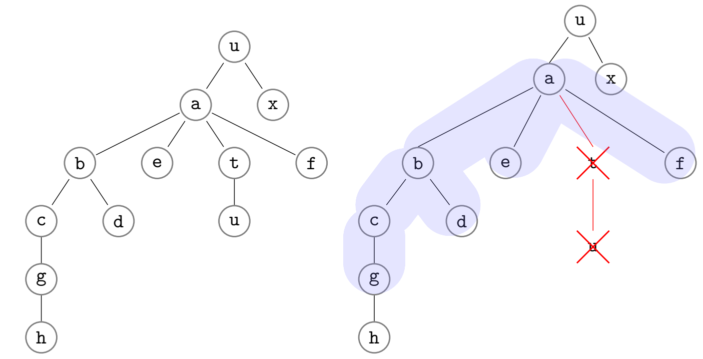
Figure 3.9: Pattern P is in T_4 at vertex a, edit distance = 0.



Tree T_5. Edited T_5 block matches P from Figure 3.5.

$$BlckDist(P, T_5) = 1;\ Editscript = Del(t).$$

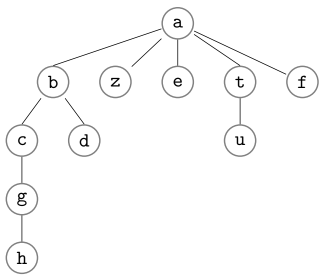
Figure 3.10: Pattern P is in T_5 at vertex a, edit distance = 1.



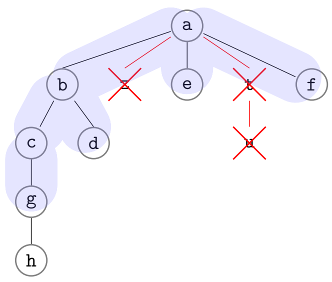
Tree T_6. Edited T_6 block matches P from Figure 3.5.

$$BlckDist(P, T_6) = 2;\ Editscript = Del(u), Del(t).$$

Figure 3.11: Pattern P is in T_6 at vertex a, edit distance = 2.



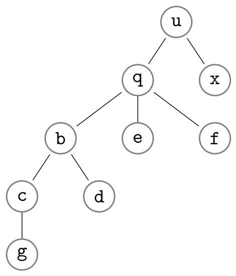
Tree T_7.



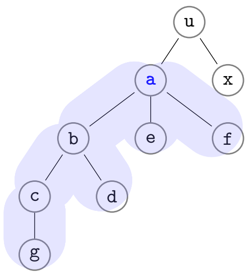
Edited T_7 block matches P from Figure 3.5.

$$BlckDist(P, T_7) = 3;\ Editscript = Del(z), Del(u), Del(t).$$

Figure 3.12: Pattern P is in T_7 at vertex a, edit distance = 3.



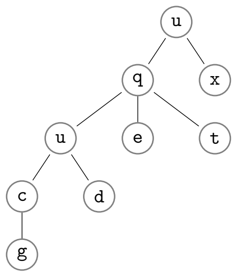
Tree T_8.



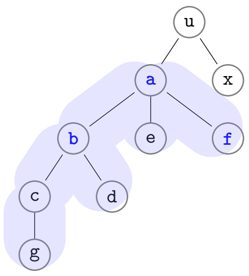
Edited T_8 block matches P from Figure 3.5.

$$BlckDist(P, T_8) = 1;\ Editscript = Rel(a).$$

Figure 3.13: Pattern P is in T_8 at vertex q, edit distance = 1.



Tree T_9.



Edited T_9 block matches P from Figure 3.5.

$$BlckDist(P, T_9) = 3;\ Editscript = Rel(a), Rel(b), Rel(f).$$

Figure 3.14: Pattern P is in T_9 at vertex q, edit distance = 3.

3.3 Pushdown Automata Approximate Block Type Matching Method

In this section, we present a new pushdown automata-based method for an approximate block type matching. The method combines the theory of pattern matching automata, 1-degree edit distance, and *Block type matching* all together.

The method consists of two phases. The first phase is to build an automaton for a specific tree pattern $P = (V_P, E_P)$. In the second phase, perform the searching via the automaton—parse an input tree $T = (V_T, E_T)$ by the automaton. A user of the method is supposed to submit only valid prefix bar notation of both the tree pattern P and the input tree T. Tree pattern P in prefix bar notation is an input for the first phase, a pushdown automaton M is the output. The second phase takes the automaton M and a searched input tree T in prefix bar notation as the input and outputs the location of all occurrences P in T.

3.3.1 Building ABTTPMA

A resulting approximate block type tree pattern matching automaton (ABTTPMA) is built together from several independent parts. We call these parts *constructions.* Each of the constructions brings another functionality to the automaton. These constructions are needed for ABTPTMA:

- Base of an automaton,
- *Block type matching* construction,
- Insert edit operation construction,
- Delete edit operation construction,
- Relabel edit operation construction.

This modularity is useful for possible customization of the resulting automaton. If any edit operation or *Block type matching* construction is not attached, the automaton works without it. However, edit distance computation cannot use the detached edit operation. Therefore, the edit distance computation is modified. For example, if the *Block type matching* is not desired, its construction can be swapped with Delete construction in the ABTTPMA. Another example is detaching the Relabel construction from ABTTPMA. Then the automaton is modified to calculate with edit distance similar to *Indel* edit distance. See Sections 3.3.1.1 — 3.3.1.6 for detailed explanation of each construction.

Pseudocode Notation Algorithm in pseudocode is presented for each of the constructions. Used methods are described here.

- `newInitialState()` A method that creates a new initial state and returns it.
- `State.addLoopTransitiion(a,b,c)` A method that adds a loop transition with label $a, b/c$ to the state itself,
- `State.add(Transition t)` A method that adds transition t. Transition t is an object with specified *to*, *from* states, cost, and label.
- `Base.add(State s)` A method that adds state s to the base of ABTTPMA.
- `prefbar(Tree t)` A method that returns t in prefix bar notation.
- `Base.getStateAt(int i)` A method that returns i^{th} state of the base of ABTTPMA.
- `createBase(Tree P)` A method that returns a base of ABTTPMA for tree P.
- `Base.addBlockTypeMatchingConstruction()` A method that adds a *Block type matching* construction to the base of ABTTPMA.
- `Base.addDeleteConstruction()` A method that adds Delete construction to the base ABTTPMA.
- `Base.addInsertConstruction()` A method that adds Insert construction to the base ABTTPMA.
- `Base.addRelabelConstruction()` A method that adds Realabel construction to the base ABTTPMA.

3.3.1.1 Base of an Automaton

Label *base* of an automaton is used for a pattern matching automaton whose input is a linearized tree pattern P in prefix bar notation. This base serves as a spine of the resulting ABTTPMA. Other automata structures are added to the base. As defined in Notation 1.1, the $\Sigma, Z_0/Z_0$ transition accepts every symbol, puts the initial stack symbol to the stack, and pops the initial symbol from the stack.

Algorithm For algorithm in pseudocode creation of the base of ABTTPMA, see Algorithm 2.

Theorem 3.3 (Correctness of Algorithm 2)**.** *Base created by Algorithm 2 works correctly (i.e., it creates the base of ABTTPMA).*

Proof. Algorithm 2 is supposed to create a pattern matching automaton. This is done by creating an initial state, and then for each symbol c of prefix bar notation P, create a new state and $c, Z_0/Z_0$ transition to it. The transitions

Algorithm 2: Algorithm for base of ABTTPMA creation.

```
Data: Tree pattern in prefix bar $prefbar(P)$
Result: Base of ABTTPMA
1  $prev = newInitialState()$ ;
2  $prev.addLoopTransition(\Sigma, \epsilon/\epsilon)$;
3  for $i = 0;\ i < prefbar(P).length();\ i++$ do
4      $c = prefbar(P).at(i)$;
5      $to = newState()$ ;
6      $prev.add(new\ Trans(to : to,\ cost : 0,\ label : c, Z_0/Z_0)$;
7      $base.add(prev)$;
8      $prev = to$;
9  end
10 return $base$;
```

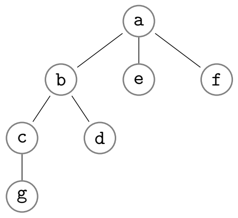


Figure 3.15: Tree pattern P.

require an empty stack. (They pop the initial stack symbol "Z_0" from the stack.) □

Theorem 3.4 (Time and space complexities of the build phase of the base of ABTTPMA)**.** *The build phase of base of ABTTPMA runs in $\mathcal{O}(|prefbar(P)|)$ and takes space of $\mathcal{O}(|prefbar(P)|)$.*

Proof. The algorithm for building the base of ABTTPMA once iterates over all symbols in prefix bar notation of P. In each iteration, there are four operations. Each takes $\mathcal{O}(1)$ time. Each iteration consists of allocating space for one transition object and one state object. Both of these allocations are in $\mathcal{O}(1)$ space. Therefore, the build phase of base of ABTTPMA runs in $\mathcal{O}(|prefbar(P)|)$ and takes space of $\mathcal{O}(|prefbar(P)|)$. □

Example 3.7. Visualization of creation of the base of an ABTTPMA.

- Consider an example tree pattern P (see Figure 3.15) :
- Prefix bar notation of P:

$$prefbar(P) = a\,b\,c\,g\,|\,|\,d\,|\,|\,e|\,f\,|\,|$$

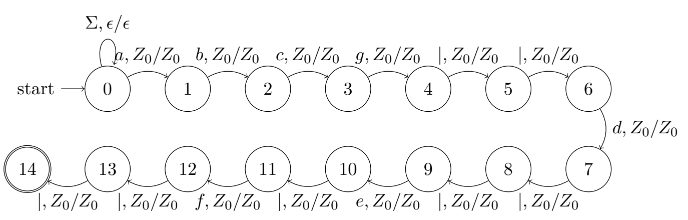


Figure 3.16: Pattern matching automaton for $prefbar(P)$.

- Pattern matching automaton for $prefbar(P)$ is shown in Figure 3.16.

Pattern matching automaton is the base of an ABTTPMA. All other constructions are attached to it or extend it.

3.3.1.2 ***Block Type Matching*** Construction

Let $P = (V_P, E_P)$ be a tree pattern and $T = (V_T, E_T)$ an input tree. *Block type matching* relaxes conditions for finding occurrences of P in T by allowing an occurrence of P in T to have some redundant subtrees. This is explained in Section 3.1. The construction of *Block type matching* works by parsing redundant subtrees of possible occurrences of a tree pattern P in an input tree T. It aims to "consume" all of the symbols of prefix bar notation of a redundant subtree. Trees in prefix bar notation have a rigid structure. Therefore, we can rely on the fact that all of the subtrees are closed by an appropriate number of closing symbols, i.e., "|". The method uses a stack of a pushdown automaton to parse each of the closing symbols for every "consumed" vertex correctly. We propose a non-deterministic structure that is to be attached to each of the locations where a redundant subtree of a P occurrence is acceptable.

Positioning The positioning of the construction is dependant on the prefix bar notation characteristics. Every two arbitrary trees share the same characteristics of prefix bar notation. Theorems 3.5, 3.6, and 3.7 clarify the conclusions arising from Definition 1.34.

Theorem 3.5 (Prefix bar notation—Descendants). *When immersing from a parent A to it's first child B, symbols "A" and "B" are concurrent in prefix bar notation.*

Theorem 3.6 (Prefix bar notation—Parent). *When going up in a tree from a child B to it's parent A, symbols "$|_B$" and "$|_A$" are concurrent in prefix bar notation if B is the right-most child of A.*

Theorem 3.7 (Prefix bar notation—Siblings). *Two sibling leaves A and B are represented by a string "$A|_A B|_B$" in prefix bar notation.*

Proof. Theorems 3.5, 3.6, 3.7 conclude from Definition 1.34. □

Let $P = (V_P, E_P)$ be a tree pattern whose occurrences are wanted to be found in an input tree $T = (V_T, E_T)$. The observation in Section 3.1 suggests a few positions where found occurrences of the tree pattern P are allowed to have redundant subtrees. These positions are specified to an arbitrary vertex $n, \forall n \in V_P$. The positions are:

1. the left-most child of $n, \forall n \in V_P$
2. the right-most child of $n, \forall n \in V_P$

Block type matching does not allow any redundant trees in between arbitrary siblings in a search pattern. Therefore, the goal is to detect the siblings in prefix bar notation. Theorem 3.5 results in Theorems 3.8 — 3.12. Theorems 3.8, 3.9, and 3.10 show where a redundant tree can be positioned in order to be parsed by *Block type matching.* Theorems 3.11, and 3.12 show where a redundant tree cannot be positioned in order to be parsed by *Block type matching.*

Theorem 3.8 (Allowed position for a redundant subtree—next to the left-most descendant)**.** *Let a and b be two vertices in an arbitrary tree T. Let prefix bar notation of the tree T be ($prefbar(T) = \ldots a\, b\, \ldots$)— symbols a and b are concurrent. Then a redundant subtree can be placed in between the vertices a and b.*

Proof. The sequence of the tree T in prefix bar notation means that the vertex b is the left-most child of a parent a. Therefore, a subtree can be placed to the left of the vertex b. □

Theorem 3.9 (Allowed position for a redundant subtree—next to the right-most descendant)**.** *Let L_1 and L_2 be two concurrent levels in a tree T. Let a be a vertex in L_1 that is a parent to a vertex $b \in L_2$. Let $|_a$ and $|_b$ be two concurrent symbols in the prefix bar notation of tree T— $prefbar(T) =, \ldots, |_b\, |_a\, \ldots,$. Then, a redundant subtree can be placed in between $|_b$ and $|_a$.*

Proof. The sequence of the tree T in prefix bar notation means that the vertex b is the right-most child of a parent a. Therefore, a subtree can be placed to the right of the vertex b. □

Theorem 3.10 (Allowed position for a redundant subtree—as a new descendant)**.** *Let a be a leave in an arbitrary tree T. Let $|_a$ be a closing symbol of the vertex a in the prefix bar notation of the tree T. Then, prefix bar notation of the tree T is $prefbar(T) = \ldots, a|_a, \ldots$) Then, a redundant subtree can be placed as a new descendant to the vertex a— in between a and $|_a$.*

Proof. The sequence of the tree T in prefix bar notation means that the vertex a is a leaf. Therefore, a subtree can be placed as a new descendant to the vertex a. $\square$

Theorem 3.11 (Disallowed position for a redundant subtree—In the beginning of prefix bar notation)**.** *Redundant subtree cannot be placed at the very beginning of the whole prefix bar notation.*

Proof. The first symbol in the prefix bar notation is always the root of an arbitrary tree T. Therefore, adding a subtree there would mean creating an entirely new tree in parallel to the T, and ABTTPMA accepts on the input only one tree. $\square$

Theorem 3.12 (Disallowed position for a redundant subtree in respect to the prefix bar representation—in between siblings)**.** *Let a and b be two vertices in an arbitrary tree T. Let $|_a$ and $|_b$ be closing symbols in the prefix bar notation of an arbitrary tree T. Let prefix bar notation of the tree T be $prefbar(T) = \ldots, |_a\, b, \ldots)$ Redundant subtree cannot be placed in between concurrent symbols $|_a$ and b.*

Proof. Sequence in the T tree prefix bar means that the vertex a is a sibling to the vertex b. *Block type matching* does not allow redundant trees in between siblings of a searched pattern. $\square$

Example 3.8. The example discusses the essence of possible positions problem. Figure 3.17 shows a sample tree pattern P whose occurrences we would search for in an input trees $T_{allowed}$ and $T_{disallowed}$. Figure 3.19 shows the input tree $T_{allowed}$ with all of the possible redundant subtrees. Each of the allowed positions is symbolized by a green triangle and labeled with a number (1—the left-most child, 2—the right-most child). If this tree is searched for tree pattern P, than *Block type matching* claims an occurrence of the tree pattern P in the input tree $T_{allowed}$. Figure 3.18 shows the input tree $T_{disallowed}$ with all of the possible disallowed positions for redundant subtrees. These positions are crossed out in the red color. If the tree $T_{disallowed}$ is searched for tree pattern P, than *Block type matching* claims no occurrence of the tree pattern P in the input tree $T_{disallowed}$.

Example 3.9. This example shows graphical visualization of the allowed and disallowed positions for redundant subtrees in input trees $T_{allowed}$ and $T_{disallowed}$ in respect to the prefix bar notation of the tree pattern P that is searched by *Block type matching.* Figure 3.17 shows tree pattern P.

- Prefix bar notation of the tree P:
  $prefbar(P) = a\, b \,|\, c\, f \,|\, g\, i \,|\,|\, h \,|\,|\, d \,|\, e \,|\,|$.

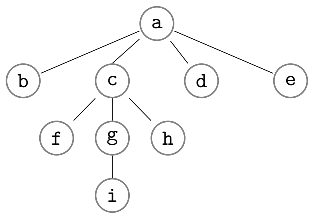


Figure 3.17: Patern tree P.

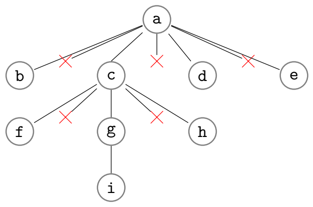


Figure 3.18: Disallowed positions for redundant subtrees in T.

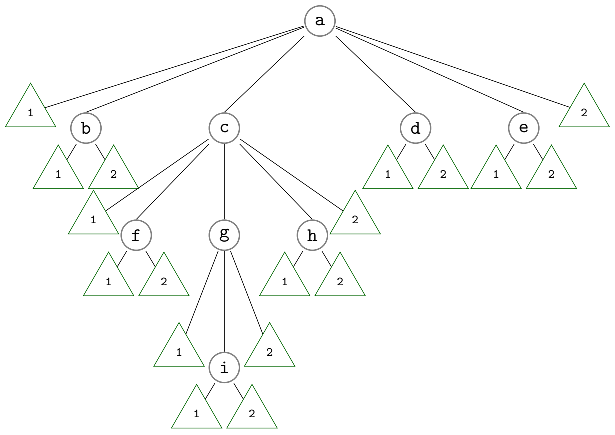


Figure 3.19: Allowed positions for redundant subtrees in T.

- Prefix bar notation of the input tree $T_{allowed}$ with highlighted allowed positions for redundant subtrees. (Prefix bar notation of an arbitrary subtree is labeled as ✓.):

  $prefbar(T_{allowed}) = a \checkmark b \checkmark | c \checkmark f \checkmark | g \checkmark i \checkmark | \checkmark | h \checkmark | \checkmark | d \checkmark | e \checkmark | \checkmark |.$

- Prefix bar notation of the input tree $T_{disallowed}$ with highlighted disallowed positions for redundant subtrees. (Prefix bar notation of an arbitrary subtree is labeled as $\times$.):

  $prefbar(T_{disallowed}) = a\, b\, | \times c\, f\, | \times g\, i\, |\, | \times h\, |\, | \times d\, | \times e\, |\, |.$

Algorithm 3: Algorithm for base extension by *Block type matching* construction.

```
Data: Tree pattern in prefix bar notation $prefbar(P)$, Base of the
      automaton $base$
Result: Base of the automaton extended by Block type matching
        construction.
for $i = 1;\ i < prefbar(P).length();\ i++$ do
    $previousChar = prefbar(P).at(i-1);$
    $char = prefbar(P).at(i);$
    if $!(prevChar == \ '|' \ \&\& \ char != \ '|')$ then
        $n = base.getStateAt(i);$
        $n.add(new\ Trans(to : n,\ cost : 0,\ label : \Sigma/\{|\}, \epsilon/|);$
        $n.add(new\ Trans(to : n,\ cost : 0,\ label : |, |/\epsilon);$
    end
end
return $base$;
```

Construction Details *Block type matching* construction behaves similarly to delete edit operation. It is needed to parse the input tree T (in prefix bar notation) without advancing in the ABTTPMA. This is done by adding two transitions. The first transition parses anything but the unique symbol | from the input and puts the special symbol to the stack. While the second transition does the exact opposite, it parses only the special symbol "|" and pops one from the stack. The construction is presented in Figure 3.20.

Algorithm For algorithm in pseudocode for attaching a *Block type matching* construction construction see Algorithm 3.

Theorem 3.13 (Correctness of Algorithm 3)**.** *Block type matching construction created by Algorithm 3 works correctly, i.e., it builds Block type matching construction to parse redundant trees.*

Proof. Algorithm 3 is supposed to create a construction that parses a prefix bar notation of redundant subtree X. The prefix bar notation of X consists of symbols from the alphabet and the special symbol "|". In prefix bar notation, each symbol (opening symbol of a subtree) is in pair with the "|" (closing symbol of a subtree). Whenever a loop transition $\Sigma/\{|\}, \epsilon/|$ is added a transition $|, |/\epsilon$ is added too (see lines 6 and 7). Therefore, for every opening symbol, a closing symbol is parsed as well. Both of the transitions operate with the stack of an automaton by putting a mark for every opening symbol, respectively deleting one for every closing symbol. Therefore, only pairs of a symbol and the closing symbol can be parsed. Valid location for a *Block*

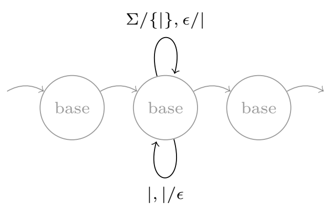


Figure 3.20: Construction to assure *Block type matching.*

Type matching construction is described in Section 3.3.1.2. Shortly, the forbidden position is between symbols "|" and "a", where a is any symbol from the alphabet. Condition on line 4 checks it. □

Theorem 3.14 (Time and space complexities of the build phase of *Block type matching* construction)**.** *Build phase of Block type matching construction runs in $\mathcal{O}(|prefbar(P)|)$ and takes space of $\mathcal{O}(|prefbar(P)|)$.*

Proof. Algorithm for building *Block type matching* construction once iterates over all symbols in prefix bar notation of P. If the condition is met (line 4) there are three operations to be executed. Each takes $\mathcal{O}(1)$ time. In these three operations, there are two alocations of space for a transition object. Both of these allocations are in $\mathcal{O}(1)$ space. Therefore, the build phase of delete construction runs in $\mathcal{O}(|prefbar(P)|)$ and takes space of $\mathcal{O}(|prefbar(P)|)$. □

Example 3.10. This example illustrates the design of the construction. See Figure 3.20. It consists of two-loop transitions, highlighted in the black color. These are added to the base of an automaton. The base is in the gray color.

3.3.1.3 Shared Features of Edit Operations Constructions

This section discusses the shared features of constructions of all edit operations. These constructions are based on the approach presented by Melichar in [34]. However, there are new requirements for an ABTTPMA because of parsing trees.

- Trees are in prefix bar notation on the input.
- Only leaves can be deleted or inserted.
- Approach from Section 2.2.1 is used.

There is a need to have some counter that should be incremented to indicate the number of edit operations while the constructions handle the edit operation. The counter feature is implemented via multiple separate levels. In the same manner as in Section 2.2.1. Each level indicates different (increasing by one) value of edit distance. As ABTTPMA is created for a maximal edit distance—specific number k, it is always possible to pre-generate an automaton with $k+1$ levels. (The first level is the base of the automaton + k different levels for each edit distance value.) Each of the three edit operation constructions starts by expanding the base (see Section 3.3.1.1) with additional k levels of an automaton. Then they add various transitions between these levels.

3.3.1.4 Delete Edit Operation Construction

Let $P = (V_P, E_P)$ and $T = (V_T, E_T)$ be two labeled unranked rooted ordered trees. Occurrences of a tree pattern P are searched for in an input tree T. T is an input tree we search in for occurrences of a tree pattern P. The search is conducted with *Block type matching* specifications. Delete edit operation deletes a leaf or multiple leaves that form a subtree in the input tree T. The idea of the Delete edit operation is that there are redundant leaves in the input tree T. Therefore, there is a need to parse these redundant leaves without effectively advancing in the pattern matching automaton and increment by one the sum of edit operations for each deleted vertex.

Positioning Delete edit operation is applied wherever *Block type matching* can not be applied, but at the first state of the base, see Theorems 3.11, and 3.12. Transitions described below are added to every state of an ABTTPMA without *Block type matching* construction.

Construction Details The parsing of a redundant tree is done by construction that is created similarly as a *Block type matching* construction. Also, the counter is implemented here. See detailed steps:

1. Create k new levels $(L_1, \dots, L_k)$ by cloning the base of an automaton.
2. Repeat the following until there are no appropriate vertices in level L_1:
   a) Find a vertex n_x at position x that is not eligible for *Block Type matching* construction on level L_1, $n_{x,1} \in L_1$.
      i. Connect $n_{x,1}$ with the vertex $n_{x,2} \in L_2$ (the same position in the next level).
      ii. The connection is a new transition for every symbol except the one used to hop on to the state to the right of the vertex at position x. Symbol "$|$" is added to the automaton stack.
      iii. Add a loop $|, |/\epsilon$ transition to a vertex n_2 that pops a symbol "$|$" from the automaton stack.
   b) Repeat the second step k-times (for every pair of levels.) $\rightarrow$ connect all pairs of vertices at position x $((n_{x,1}, n_{x,2}), \dots, (n_{x,k-1}, n_{x,k}))$.

Algorithm 4: Algorithm for base extension by delete construction.

```
Data: Tree pattern in prefix bar $prefbar(P)$, Base of the automaton $base$
Result: Base of the automaton extended by delete construction.
1  for $i = 1;\ i < prefbar(P).length();\ i++$ do
2      $previousChar = prefbar(P).at(i - 1);$
3      $char = prefbar(P).at(i);$
4      if $prevChar == \ '|' \ \&\& \ char! = \ '|'$ then
5          $n = base.getStateAt(i);$
6          $n.add(new\ Trans(to : n,\ cost : 1,\ label : \Sigma/\{|\}, \epsilon/|);$
7          $n.add(new\ Trans(to : n,\ cost : 0,\ label : |, |/\epsilon);$
8      end
9  end
10 return $base$;
```

Algorithm For algorithm in pseudocode for attaching a delete construction, see Algorithm 4.

Theorem 3.15 (Correctness of Algorithm 4)**.** *Delete construction created by Algorithm 4 works as intended, i.e., it builds Delete construction to parses redundant trees and increment the edit distance counter.*

Proof. Algorithm 4 is supposed to create a construction that parses a prefix bar notation of a tree X. The prefix bar notation of X consists of symbols and the special character "|". In prefix bar notation, each symbol (opening symbol of a vertex) is in pair with the "|" (closing symbol of a vertex). Whenever a loop transition $\Sigma/\{|\}, \epsilon/|$ is added a transition $|, |/\epsilon$ is added too (see lines 6 and 7). Therefore, for every opening symbol, a closing symbol is parsed as well. Both of the transitions operate with the stack of an automaton by putting a mark for every opening symbol, respectively deleting one for every closing symbol. Therefore, only pairs of a symbol and the closing symbol can be parsed. Each deleted leaf/subtree X (multiple nested leaves form a subtree) has it's cost $cost_X$. This value is a sum of all vertices of X. Therefore, for every opening symbol, the cost of value one is added to the transition signature. Valid location for delete construction concludes from the fact that a set of states with *Block type matching* construction and a set of states with delete construction are disjoint. Section 3.3.1.2 describes positions where *Block type matching* is not allowed. Therefore, these positions are exactly the ones where delete construction is attached. Condition on line 4 checks for the specified position (negation of condition used in *Block type matching*). □

Theorem 3.16 (Time and space complexities of the build phase of delete construction)**.** *Build phase of delete construction runs in $\mathcal{O}(|prefbar(P)|)$ and takes space of $\mathcal{O}(|prefbar(P)|)$.*

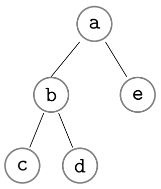


Figure 3.21: Tree P used for Delete construction example.

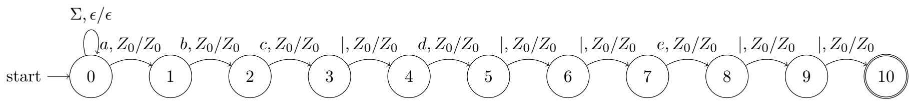


Figure 3.22: The base of an ABTTPMA for P.

Proof. Algorithm for building delete construction once iterates over all symbols in prefix bar notation of P. If the condition is met (line 4), there are three operations to be executed. Each takes $\mathcal{O}(1)$ time. In these three operations, there are two allocations of space for a transition object. Both of these allocations are in $\mathcal{O}(1)$ space. Therefore, the build phase of delete construction runs in $\mathcal{O}(|prefbar(P)|)$ and takes space of $\mathcal{O}(|prefbar(P)|)$. □

Example 3.11 (Delete edit operation construction)**.** This examples demonstrates idea presented in Section 3.3.1.4.

- Sample input that is to be extended with Delete construction.
  - Input paramater k: max edit distance.

$$k = 3$$

  - The tree pattern P is considered, see Figure 3.21.
  - Tree P in prefix bar notation:

$$prefbar(P) = a\,b\,c\,|\,d\,|\,|\,e\,|\,|$$

  - The base of an automaton for tree P, see Figure 3.22.

- Construction details

  The example follows instructions in Section 3.3.1.4.

  - Create k new levels $(L_1, \ldots, L_k)$, see Figure 3.23.
  - Find a vertex n_x at position x that is not eligible for *Block Type matching* construction on level L_1, $n_{x,1} \in L_1$, see Figure 3.24.

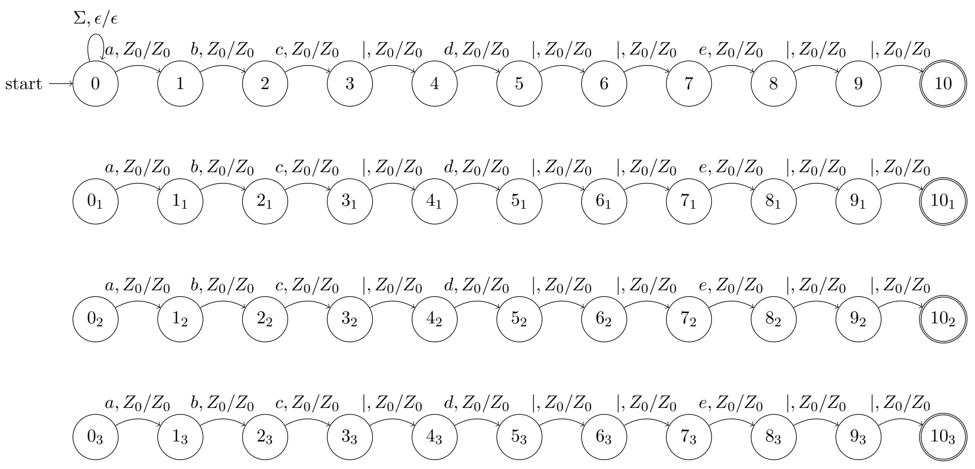


Figure 3.23: k new levels added to the automaton of the tree pattern P.

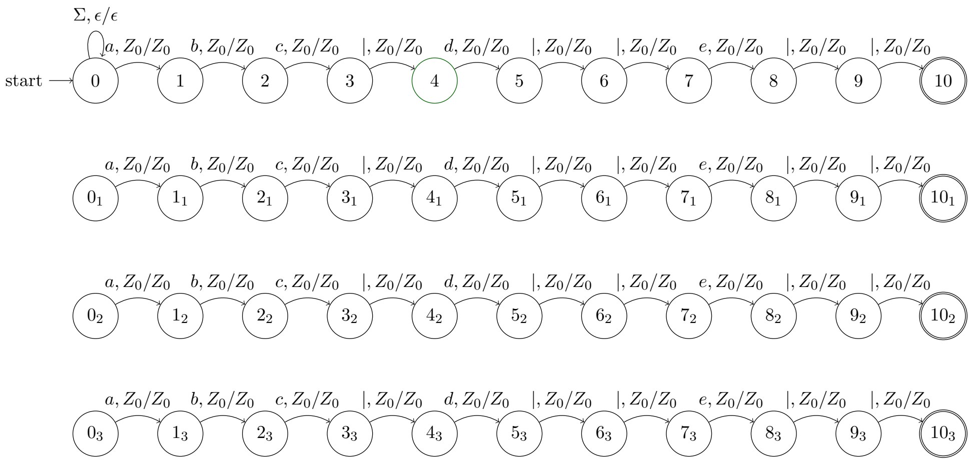


Figure 3.24: Vertex n_x at position x that is not eligible for *Block Type matching* construction on level L_1, $n_{x,1} \in L_1$.

- Added transitions for vertices at position x on adjacent levels, see Figure 3.25.
- Apply previous steps to all vertices not eligible for *Block type matching* construction. see Figure 3.26.

We are aware of some unusable redundant states and transitions that are produced by this idea. This problem is addressed in the implementation of the method.

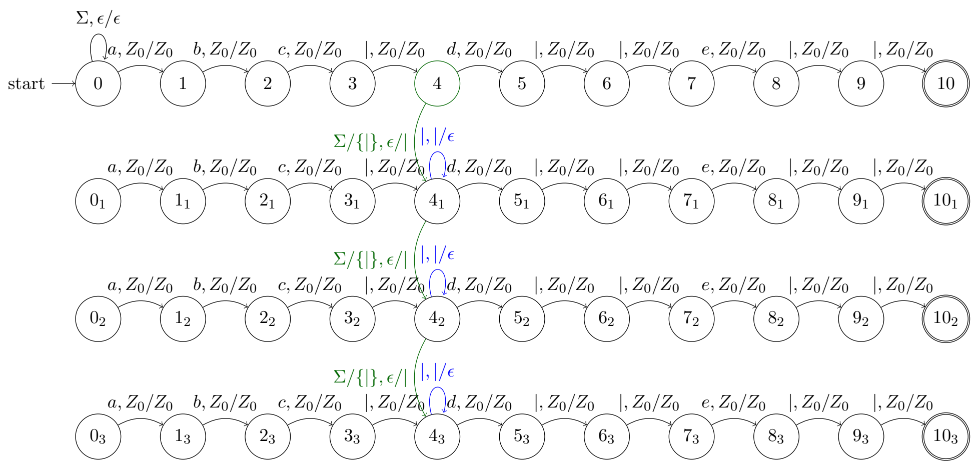


Figure 3.25: Added transitions for vertices at position x on adjacent levels.

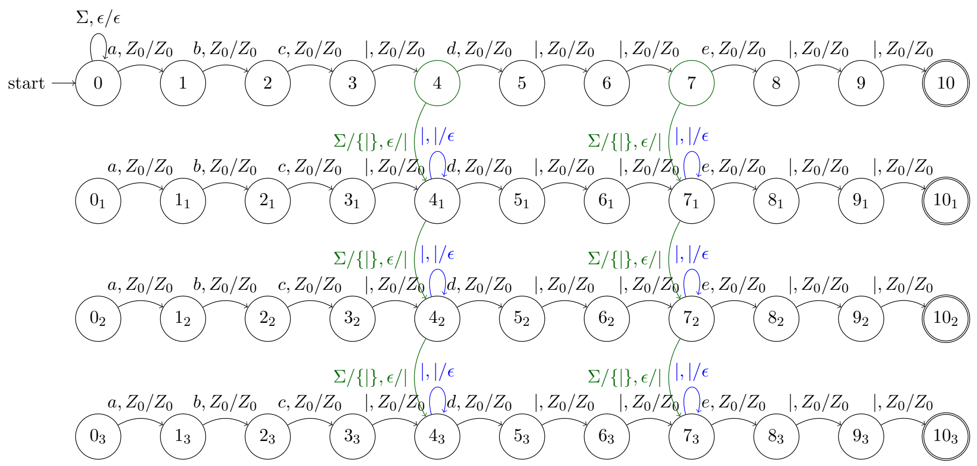


Figure 3.26: Applied previous steps to all vertices not eligible for *Block type matching* construction.

3.3.1.5 Insert Edit Operation Construction

Let $P = (V_P, E_P)$ and $T = (V_T, E_T)$ be two labeled unranked rooted ordered trees. Occurrences of a tree pattern P are searched for in an input tree T . T is an input tree we search in for occurrences of a tree pattern P via *Block type matching*. Insert edit operation inserts a leaf or multiple leaves (these leaves form a subtree) into the input tree T. The idea of Insert edit operation is that there are missing leaves (or subtrees) in the input tree T. The Insert construction is added to the base of an ABTTPMA created from the tree pattern P. As leaves are missing in the input tree T (they are to be inserted), there is a need to advance in the ABTTPMA without parsing the input tree

T. Of course, the counter has to be incremented for every added leaf. These ideas suggest the following:

- $\epsilon, Z_0/Z_0$ transitions are used to advance further in the automaton without parsing the input tree T.
- Implement edit distance counter via multiple levels of the automaton. Each level represents a value of edit distance.

Positioning Let S be an arbitrary subtree whose total number of vertices is x, where $1 \leq x \leq k$ (k — maximal edit distance). S can be inserted into an input tree T. There has to be an $\epsilon, Z_0/Z_0$ transition for each S in an ABTTPMA. As the Insert edit operation recursively inserts only leaves, every subtree (made up by those leaves) in the prefix bar is written in one block. Therefore, for every subtree, there is only one $\epsilon, Z_0/Z_0$ transition. Let $P = (V, E)$ be a tree pattern. Let n_1 be a node, $n_1 \in V$.

Let B be a subtree whose root is n_1. If $1 \leq numberOfVertices(B) \leq k$, than there should be an transition:

- from the state $a \in L_x$, where a has the outgoing transition with label $n_1, Z_0/Z_0$.
- to a state $b \in L_{x+numberOfVertices(B)}$, where b has incoming transition with label $|_{n_1}, Z_0/Z_0$.

Construction Details The insert edit operation is done by construction that is created in the following manner:

1. Create k new levels $(L_1, \ldots, L_k)$ by cloning the base of an automaton. Each level represents an edit distance value.
2. Search for subtrees with the number of descendants less than k. $\rightarrow$ Search backwards the prefix bar notation of P for symbols "|".
3. For every "|" search for all nested "|". These nested "|" symbols represent nested vertices (these form a subtree). Find all subtrees with the number of vertices up to the value of k.

   For example:

   - Sequence "$x_1\,|$", $x_1 \in \Sigma$ is a subtree with one vertex.
   - Sequence "$x_1\,x_2\,|\,|$", $x_1 \in \Sigma$ is a subtree with two vertices.
   - Sequence "$x_1\,x_2\,x_3\,|\,|\,|$", $x_1 \in \Sigma$ is a subtree with three vertices.
   - Sequence "$x_1\,x_2\,|\,x_3\,|\,|$", $x_1 \in \Sigma$ is a subtree with two siblings and a parent.

Algorithm 5: Algorithm for base extension by insert construction.

```
Data: Tree pattern in prefix bar notation $prefbar(P)$, Base of the
      automaton $base$, Max edit distance $k$
Result: Base of the automaton extended by insert construction.
for $i = prefbar(P).length() - 1; i \geq 0; i--$ do
    if $prefbar(P).at(i) == '|'$ then
        $depth = 1$;
        $cost = 1$;
        for $j = i - 1; j \geq 0; j--$ do
            if $prefbar(P).at(j) == '|'$ then
                $depth++$;
                $cost++$;
            end
            else
                $depth--$;
            end
            if $cost > k$ then
                $break$;
            end
            if $depth == 0$ then
                $to = base.getStateAt(i)$;
                $from = base.getStateAt(j)$;
                $from.add(new\ Trans(to : to,\ cost : cost,\ label :$
                  $\epsilon, Z_0/Z_0$;
            end
        end
    end
end
return $base$;
```

4. Add $\epsilon, Z_0/Z_0$ transition for every missing subtree. The transition starts from a state just before the missing subtree on the level L_0 and goes to the vertex just after the subtree on the level L_x where x is a number of vertices of the missing subtree.

5. Clone all the transitions created from the previous step to other levels.

Algorithm For algorithm in pseudocode for attaching an insert construction, see Algorithm 5.

Theorem 3.17 (Correctness of Algorithm 5). *Insert construction created by Algorithm 5 works correctly, i.e., it creates Insert construction to simulate parsing of an arbitrary tree and increment the edit distance counter.*

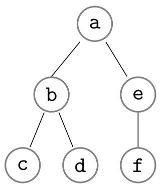


Figure 3.27: Tree P used for Insert construction example.

Proof. Algorithm 5 is supposed to create a construction that simulates parsing of an arbitrary tree X when such a subtree is missing. This is done by adding an $\epsilon, Z_0/Z_0$ transition to bypass block of states where the X would be parsed. The prefix bar notation of a missing subtree is always in one block of symbols. See Definition 1.34. Insert edit operation recursively adds leaves. Therefore, only the whole subtrees of P can be considered for insertation. The algorithm computes positions of all of the possible subtrees of the tree pattern P whose number of vertices is up to maximal edit distance. For every closing symbol "|" (line 2), its opening symbol is found (line 16)—beginning and end of a new transition. If the cost is at most maximal edit distance (line 13), a transition is created (line 19). It requires an empty stack. (Pops the initial stack symbol "Z_0" from the stack.) □

Theorem 3.18 (Time and space complexities of the build phase of insert construction)**.** *Build phase of insert construction runs in $\mathcal{O}(|prefbar(P)|^2)$ and takes space of $\mathcal{O}(|prefbar(P)|^2)$.*

Proof. The algorithm for building Insert construction iterates over all symbols in prefix bar notation of P. If the condition is met (line 5), a new iteration starts. This iteration has $\mathcal{O}(|prefbar(P)|)$ steps. Inside the inner iteration, if conditions are met, there are five operations to be executed. Each operation runs in $\mathcal{O}(1)$ time. In these five operations, there is only one allocation of space for transition object. Therefore, the build phase of insert construction runs in $\mathcal{O}(|prefbar(P)|^2)$ and takes space of $\mathcal{O}(|prefbar(P)|^2)$. □

Example 3.12 (Insert edit operation construction)**.** This examples demonstrates idea presented in Section 3.3.1.5.

- Sample input that is to be extended with Insert construction.
  - Input paramater k: max edit distance.

$$k = 3$$

  - The tree pattern P is considered, see Figure 3.27.

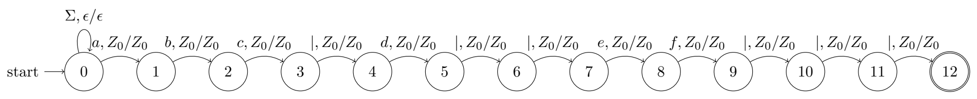


Figure 3.28: The base of an ABTTPMA for P.

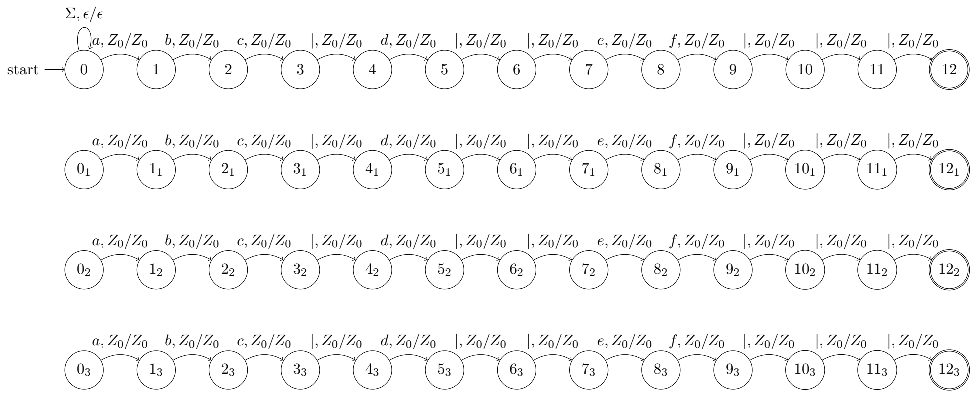


Figure 3.29: k new levels added to the automaton of the tree pattern P.

- Tree P in prefix bar notation:

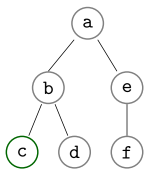
$$prefbar(P) = a\,b\,c\,|\,d\,|\,|\,e\,f\,|\,|\,|$$

- The base of an automaton for tree P, see Figure 3.28.

• Construction details

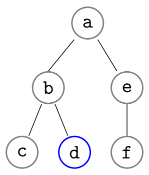
The example follows instructions in Section 3.3.1.5.

- Create k new levels $(L_1, \ldots, L_k)$, see Figure 3.29.
- Search for subtrees with the number of vertices up to k, see Figures 3.30, 3.31, 3.32, 3.33, and 3.34.



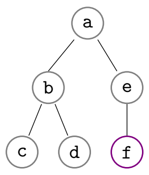
$prefbar(P) = a\,b\,\mathbf{c}\;|\,d\,|\,|\,e\,f\,|\,|\,|.$

Figure 3.30: Found subtree 1.



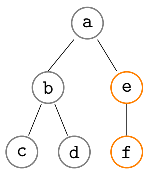
$prefbar(P) = a\,b\,c\,|\,\mathbf{d}\;|\,|\,e\,f\,|\,|\,|.$

Figure 3.31: Found subtree 2.



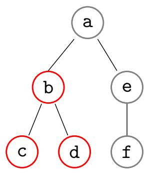
$prefbar(P) = a\,b\,c\,|\,d\,|\,|\,e\,\mathbf{f}\;|\,|\,|.$

Figure 3.32: Found subtree 3.



$prefbar(P) = a\,b\,c\,|\,d\,|\,|\,\mathbf{e}\;\mathbf{f}\;|\;|\;\;|.$

Figure 3.33: Found subtree 4.



$prefbar(P) = a\,\mathbf{b}\;\mathbf{c}\;|\;\mathbf{d}\;|\;|\,e\,|\,|.$

Figure 3.34: Found subtree 5.

- Add $\epsilon, Z_0/Z_0$ transition for every missing subtree, see Figure 3.35.
- Apply previous steps to all levels where applicable. See Figure 3.36.

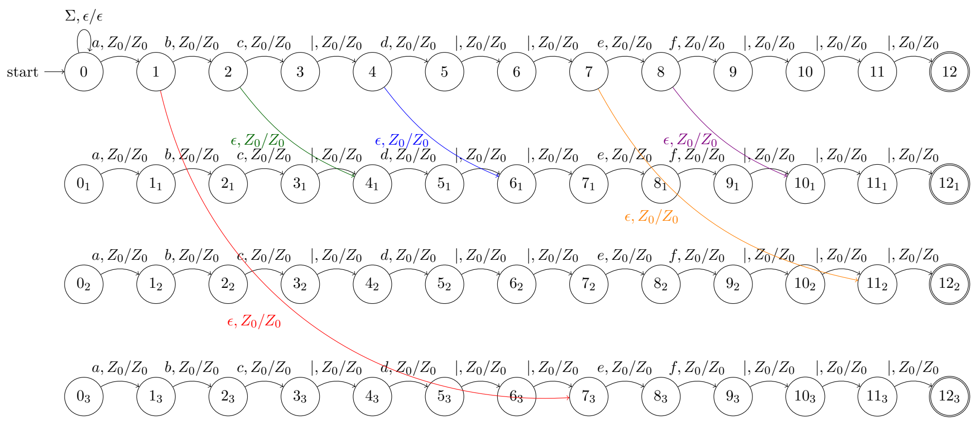


Figure 3.35: Added transitions for every missing subtree.

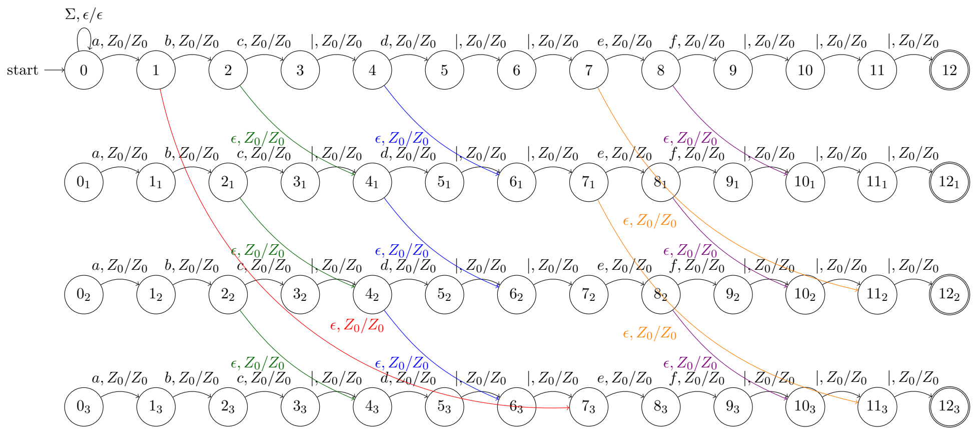


Figure 3.36: Added transitions for every missing subtree.

We are aware of some unusable redundant states and transitions that are produced by this idea. This problem is addressed in the implementation of the method.

3.3.1.6 Relabel Edit Operation Construction

Let $P = (V_P, E_P)$ and $T = (V_T, E_T)$ be two labeled unranked rooted ordered trees. Occurrences of a tree pattern P are searched for in an input tree T. T is an input tree we search in for occurrences of a tree pattern P via *Block type matching*. Relabel edit operation relabels a vertex in the input tree T. The relabel operation solves a problem where there is a correctly placed vertex (nodes) but with a wrong label. This means that the inner structure of the tree is correct, only badly labeled. Therefore, the idea is to parse the wrong label and increment the edit distance counter. There is no need to manipulate

Algorithm 6: Algorithm for base extension by relabel construction.

```
Data: Tree pattern in prefix bar notation $prefbar(P)$, Base of the
      automaton $base$
Result: Base of the automaton extended by relabel construction.
$prev = base.getRoot()$ ;
for $i = 0;\ i < prefbar(P).length();\ i++$ do
    $to = base.getStateAt(i + 1)$;
    if $prefbar(P).at(i)\ != \ '|'$ then
        $prev.add(new\ Trans(to : to,\ cost : 1,\ label : \Sigma/\{|\}, Z_0/Z_0)$;
    end
    $prev = to$;
end
return $base$;
```

with "|" symbols (these only handle the structure of a tree). This is done by creating new k levels, where k is the maxim edit distance, and adding transitions on $\Sigma/\{|\}, Z_0/Z_0$ level L_i to L_{i+1}.

Positioning These transitions are added to all states of an ABTTPMA whose base transition is not set to label $|, Z_0/Z_0$.

Construction Details These transitions are created in the following way.

1. Create k new levels $(L_1, \ldots, L_k)$ by cloning the base of an automaton. Each level represents an edit distance value.

2. Search for all states in the automaton that does not have transition on label $|, Z_0/Z_0$

3. For each of them, add a transition on $\Sigma, Z_0/Z_0$ to the next lower level and one state further.

4. Clone all the transitions created from the previous step to other levels where possible.

Algorithm For algorithm in pseudocode for attaching a relabel construction, see Algorithm 6.

Theorem 3.19 (Correctness of Algorithm 6)**.** *Relabel construction created by Algorithm 6 works as intended, i.e., it builds Relabel construction to simulate the relabeling of a vertex and increment the edit distance counter.*

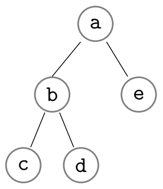


Figure 3.37: Tree P used for Relabel construction example.

Proof. Algorithm 6 is supposed to create a construction that simulates the relabeling of a vertex. The structure of a tree is left intact. Relabel construction consists of a transition with the cost of value one that is added to every state of the base of ABTTPMA whose base transition label is a symbol (line 4). See Section 3.3.1.6. Algorithm 6 loops through all states and add a transition (line 5) to those that meet the condition (line 4). Added transitions simulate relabeling in an input tree. These transitions parse any symbol (line 5). It requires an empty stack. (Pops the initial stack symbol "Z_0" from the stack.) □

Theorem 3.20 (Time and space complexities of the build phase of relabel construction)**.** *Build phase of relabel construction runs in* $\mathcal{O}(|prefbar(P)|)$ *and takes space of* $\mathcal{O}(|prefbar(P)|)$*.*

Proof. Algorithm for building relabel construction once iterates over all symbols in prefix bar notation of P. If the condition is met (line 4), there is one allocation of space for a transition object. Therefore, the build phase of delete construction runs in $\mathcal{O}(|prefbar(P)|)$ and takes space of $\mathcal{O}(|prefbar(P)|)$. □

Example 3.13 (Relabel edit operation construction)**.** This examples demonstrates idea presented in Section 3.3.1.6.

- Sample input that is to be extended with Relabel construction.
  - Input paramater k: max edit distance.

$$k = 3$$

  - The tree pattern P is considered, see Figure 3.37.
  - Tree pattern P in prefix bar notation:

$$prefbar(P) = a\, b\, c\, |\, d\, |\, |\, e\, |\, |$$

  - The base of an automaton for tree P, see Figure 3.38.
- Construction details

  The example follows instructions in Section 3.3.1.6.

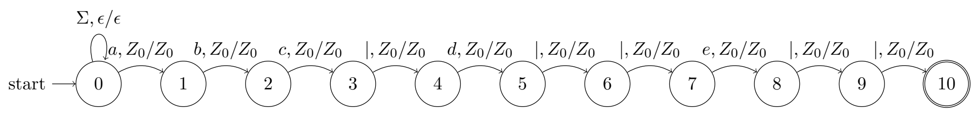


Figure 3.38: The base of an ABTTPMA for P.

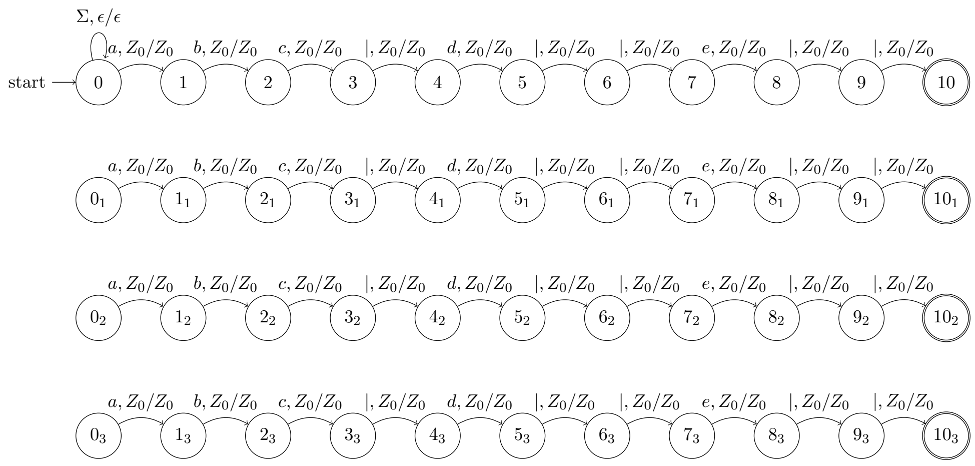


Figure 3.39: k new levels added to the automaton of the tree pattern P.

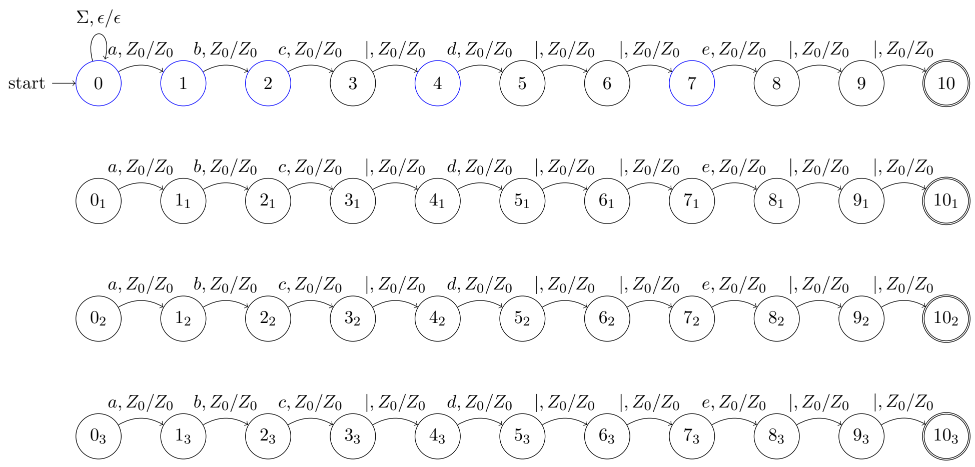


Figure 3.40: All states of L_1 of the automaton without outgoing transition $|, Z_0/Z_0$.

- Create k new levels $(L_1, \ldots, L_k)$, see Figure 3.39.
- Search for all states of L_1 of the automaton without outgoing transition on a $|, Z_0/Z_0$ label, see Figure 3.40.
- Added transitions to the found states, see Figure 3.41.

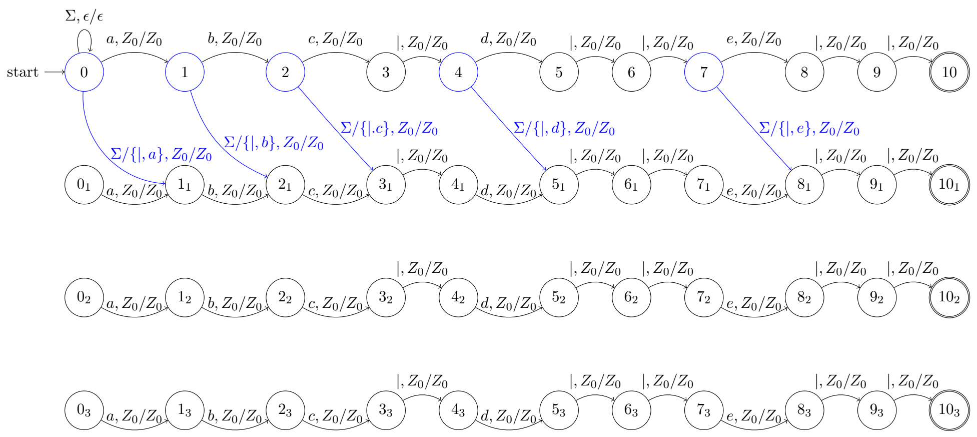


Figure 3.41: Added transitions to the found states.

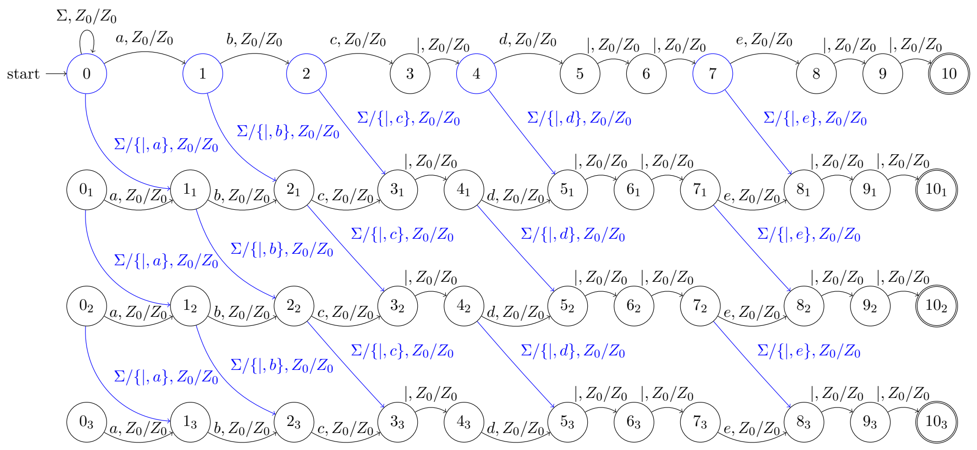


Figure 3.42: Applied previous steps to all levels where applicable.

– Apply previous steps to all levels, see Figure 3.42.

We are aware of some unusable redundant states and transitions that are produced by this idea. This problem is addressed in the implementation of the method.

3.3.1.7 Complete ABTTPMA Construction

Here we present the whole process of building an ABTTPMA. The process consists of creating the base of ABTTPMA, adding *Block type matching* construction, and all three edit operation constructions. This modularity has an advantage of possible customization of ABTTPMA by omitting any construction.

Algorithm 7: Algorithm for ABTTPMA.

```
Data: Tree pattern in prefix bar $prefbar(P)$
Result: ABTTPMA
1 Base $base$ = createBase ($P$);
2 $base$.add$BlockTypeMatching$Construction();
3 $base$.addDeleteConstruction();
4 $base$.addInsertConstruction();
5 $base$.addRelabelConstruction();
6 return $base$;
```

Algorithm For algorithm in pseudocode for ABTTPMA, see Algorithm 7.

Theorem 3.21 (*Block type matching* and delete construction's function do not affect each other)**.** *Each state from ABTTPMA can attach either Block type matching construction, or delete construction.*

Proof. Let B and D be a set of states where Block type matching construction, respectively delete construction is attached. Set B is disjoint to set D and vice-versa, see Sections 3.3.1.4. Therefore, in ABTTPMA two states cannot have both Block type matching construction, and delete construction. Insert construction, Relabel construction, and base transitions require empty stack—they pop the initial stack symbol "Z_0" from the stack. Therefore, *Block type matching* and delete constructions cannot alter their stack operations between each other. □

Theorem 3.22 (Construction's functions do not influence each other)**.** *Each symbol $c \in prefbar(T)$ is handled only by one construction.*

Proof. Block type matching construction function cannot be affected by delete construction, see Theorem 3.21. Let $T = (V, E)$ be an arbitrary labeled unranked rooted ordered tree with the root vertex r. The prefix bar notation of T is written in one block. The first symbol of prefix bar notation of T is r, and the last one is $|_r$-closing symbol of r. *Block type matching* construction cannot be affected by any other construction or base transition as these require an empty stack. The stack is empty only if *Block type matching* construction parses an equal number of opening symbols and closing symbols "|". Therefore, *Block type matching* construction must parse the whole block of prefix bar notation of an arbitrary subtree before any other construction or base transition can be used.

In the same manner, as *Block type matching* construction delete edit construction function is not affected by any other construction or base transition. Therefore, delete construction must parse the whole block of a prefix bar notation of an arbitrary subtree before any other construction or base transition can be used.

Insert construction inserts subtrees by precomputed transitions that bypass a whole block of transitions that correspond to a whole block of the prefix bar notation of an arbitrary subtree. It requires an empty stack. (Pops the initial stack symbol "Z_0" from the stack.) The function of relabel construction only changes a label; the structure of an arbitrary tree is always untouched. Transitions of the base of ABTTPMA correspond to the prefix bar notation of the tree pattern. Every transition requires an empty stack. (Pop the initial stack symbol "Z_0" from the stack.) Therefore, adding all constructions to ABTTPMA does not create a risk of constructions being influenced by each other. □

Theorem 3.23 (Correctness of Algorithm 7)**.** *ABTTPMA created by Algorithm 7 works as intended, i.e., it builds ABTTPMA.*

Proof. On the first line, the algorithm creates the base of ABTTPMA. On the next lines, all of the constructions are added to the base. The functions of constructions are not influenced by each other. See Theorem 3.22. □

Theorem 3.24 (Time and space complexities of build phase of ABTTPMA)**.** *Build phase of ABTTPMA runs in $\mathcal{O}(|prefbar(P)|^2)$ and takes space of $\mathcal{O}(|prefbar(P)|^2)$.*

Proof. Algorithm for building ABTTPMA calls build operation of each construction once. These operations are called sequentially. The most both time and space consuming is insert construction. It runs in $\mathcal{O}(|prefbar(P)|^2)$ takes space of $\mathcal{O}(|prefbar(P)|^2)$. Therefore, the build phase ABTTPMA runs in $\mathcal{O}(|prefbar(P)|^2)$ and takes space of $\mathcal{O}(|prefbar(P)|^2)$. □

Example 3.14. The example presents the build phase of ABTTPMA. A small tree pattern P and parameter k set to $k = 2$ is used to make the automaton schema transparent. See Section 3.3.1 for detailed ilustrations of each construction. Every construction is highlighted in a unique color. The following coloring is used:

- *Block type matching* construction is in the black color.
- Base of an automaton and its clones (addtional levels in the automaton) are in the gray color.
- Delete edit operation is in the red color.
- Insert edit operation is in the green color.
- Relabel edit operation is in the blue color.
- Input paramater k: max edit distance.

$$k = 2$$

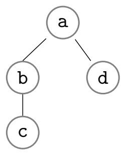


Figure 3.43: Tree P used as input for build phase of ABTTPMA.

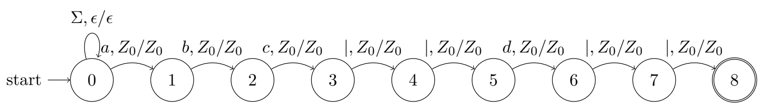


Figure 3.44: The base of an ABTTPMA for P.

- The tree pattern P is considered, see Figure 3.43.
- The tree pattern P in prefix bar notation:

$$prefbar(P) = a\,b\,c\,|\,|\,d\,|\,|$$

- The base of an automaton for tree P, see Figure 3.44.

Result Let $n = |prefbar(P)|$, $k = max\ edit\ distance$. The resulting ABTTPMA for the tree pattern P is a seven-tuple $(\bigcup_{j=0}^{k} \bigcup_{i=0}^{n} i_j, A, \{|, Z_0\}, \delta, 0_0, Z_0, \bigcup_{j=0}^{k} n_j)$, where mapping δ is shown in Table 3.1. ABTTPMA is visualized in Figure 3.45. Notation used in Table 3.1 is:

- k for a maximal edit distance,
- n for length of prefix bar notation of tree pattern P,
- A for unique symbols from $prefbar(P)$,
- $pb(X)[i]$ for an symbol at position i in prefix bar notation of an arbitrary tree X,
- $\{Subt(X) < i\}$ for a set of all subtrees $S = (V_S, E_S)$ of an arbitrary tree X where $|V_S| < i$,
- $BlckValid(v)$ for a function that returns *true* if a vertex v is suitable for *Block type matching* construction,
- $DelValid(v)$ for a function that returns *true* if a vertex v is suitable for Delete construction,
- $InsValid(v)$ for a function that returns *true* if a vertex v is suitable for Insert construction,
- $RelValid(v)$ for a function that returns *true* if a vertex v is suitable for Relabel construction.

Comment	Condition	From (start state)	Edge label	To (target state)
Base	$j \in \{0, \cdots, k\}$, $i \in \{0, \cdots, n\}$	i_j	$pb(P)[i], Z_0/Z_0$	$i + 1_j$
Block type matching 1	$j \in \{0, \cdots, k\}$, $i \in \{0, \cdots, n\}$, $BlckValid(i_j)$	i_j	$\{\|\}, \|/\epsilon$	i_j
Block type matching 2	$j \in \{0, \cdots, k\}$, $i \in \{0, \cdots, n\}$, $BlckValid(i_j)$	i_j	$\Sigma/\{\|\}, \epsilon/\|$	i_j,
Delete 1	$j \in \{1, \cdots, k\}$, $i \in \{0, \cdots, n\}$, $DelValid(i_{j-1})$	i_j	$\{\|\}, \|/\epsilon$	i_j
Delete 2	$j \in \{1, \cdots, k\}$, $i \in \{0, \cdots, n\}$, $DelValid(i_{j-1})$	i_{j-1}	$\Sigma/\{\|\}, \epsilon/\|$	i_j
Relabel	$j \in \{1, \cdots, k\}$, $i \in \{0, \cdots, n-1\}$, $RelValid(i_{j-1})$	i_{j-1}	$\Sigma\{\|, pb(P)[i]\}, Z_0/Z_0$	$i + 1_j$
Insert	$j \in \{0, \cdots, k\}$, $i \in \{0, \cdots, n-1\}$, $s \in \{Subt(P) < k\}$, $i + \|S\|_{j+\|S\|} \in Q$, $InsValid(i_j)$	i_j	$\epsilon, Z_0/Z_0$	$i + \|S\|_{j+\|S\|}$

Table 3.1: ABTTPM δ mapping.

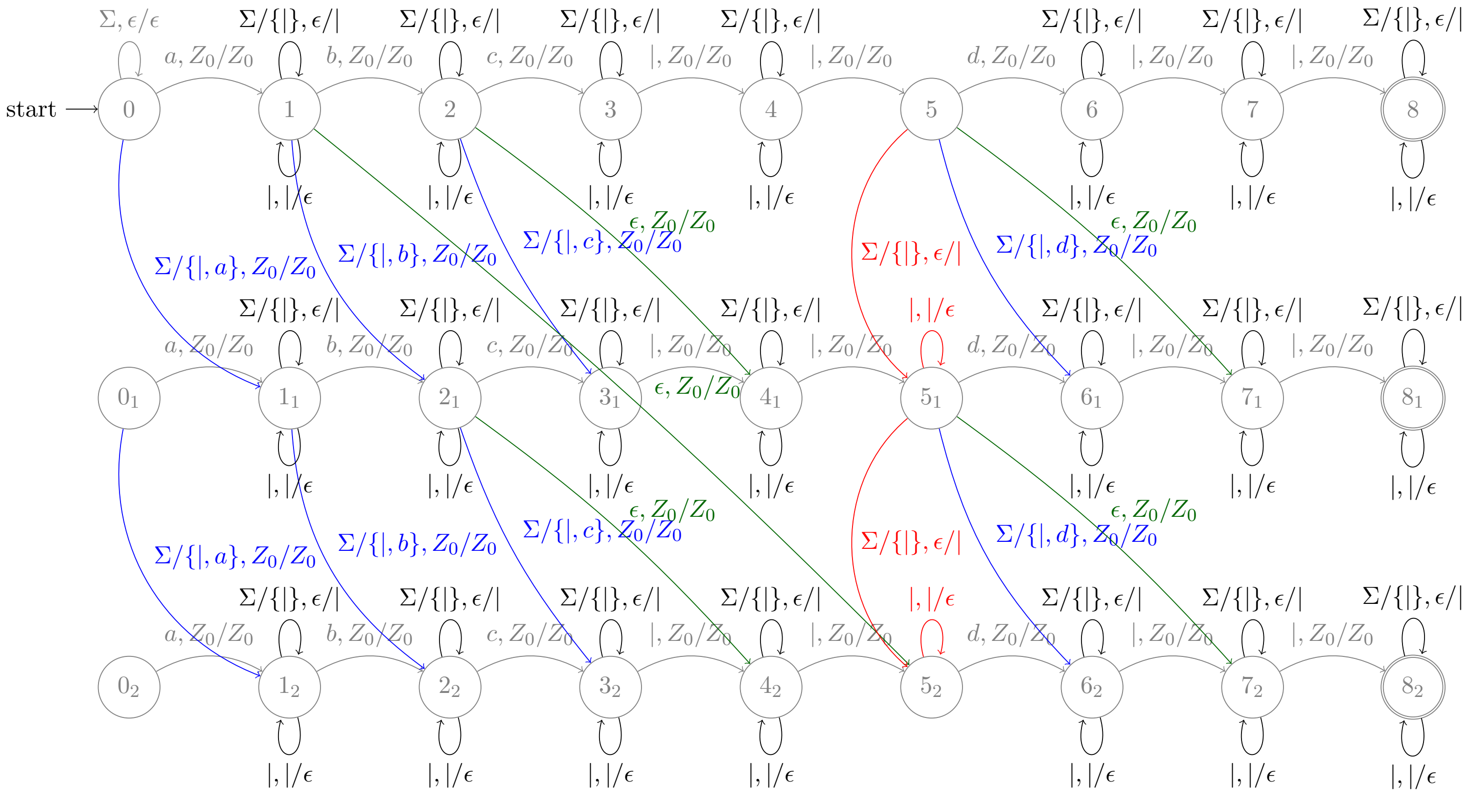


Figure 3.45: ABTTPMA for a tree pattern P.

3.3.2 Searching via ABTTPMA

ABTTPMA is a nondeterministic pushdown automaton. Therefore, using it for searching in an input tree $T = (V_T, E_T)$ for tree pattern $P = (V_P, E_P)$ is a typical parsing procedure of T by the ABTTPMA.

Theorem 3.25 (Time and space complexities of the search phase of ABTTPMA)**.** *Let $n = |prefbar(T)|$ and K is the maximal edit distance. Search phase of ABTTPMA runs in $\mathcal{O}(k \cdot n \cdot 5^k \cdot 2^{n-k})$ and takes space of $\mathcal{O}(n)$.*

Proof. Because of the Σ loop transition at state 0 of ABTTPMA, there are $\mathcal{O}(n)$ sequences (formed by all suffixes of prefix bar notation of tree pattern) to be parsed by the ABTTPMA. ABTTPMA consists of $k \cdot n$ states. For each sequence, there are k states that have up to 5 available outgoing transitions (There are five possibilities: parse, *Block type matching* operation, delete, relabel, insert.) and $n - k$ states that have up to two available outgoing transitions (There are two possibilities: parse, *Block type matching* operation.). Therefore, the time complexity is $\mathcal{O}(k \cdot n \cdot 5^k \cdot 2^{n-k})$.

ABTTPMA uses its stack. Up to one item is added to the stack for each opening symbol in the input sequence. Therefore there can be up to $\mathcal{O}(n)$ items in the stack. Therefore, the space complexity is $\mathcal{O}(n)$ □

3.3.3 ABTTPMA Determinization

Some pushdown automaton belong to pushdown automata classes that consist of determinizable automata, see Observation 1.2.4. These classes of pushdown automata are :

- Input driven
- Visibly pushdown
- Height deterministic

Theorem 3.26 (ABTTPMA determinization 1)**.** *ABTTPMA does not belong to Input driven class.*

Proof. The idea of input driven pushdown automata is that the input symbols uniquely determine whether the automaton pushes a symbol, pops a symbol, or leaves the pushdown unchanged. This does not apply to ABTTPMA. For example, the state with label 1 in Figure 3.45 for symbol a has two transitions with different stack operation—*Block type matching* loop transition adds "|" symbol to the stack or the base transition does not modify the stack. □

Theorem 3.27 (ABTTPMA determinization 2)**.** *ABTTPMA does not belong to Height deterministic class.*

Proof. Class of height deterministic pushdown automata consists of automata that, for any given input string, the stack height during any (nondeterministic) computation is a priori fixed. This does not apply to ABTTPMA. Up to one item is added to the stack for each opening symbol in the input sequence. Therefore, there can be up to $\mathcal{O}(n)$ items in the stack. □

Theorem 3.28 (ABTTPMA determinization 3)**.** *ABTTPMA belongs to Visibly pushdown class.*

Proof. Visibly Pushdown automaton is a pushdown automaton whose stack operations are determined by the input symbol it reads. Nested word is a linear structure with a nesting relation formed by associating open-symbols (i.e., $\Sigma/\{|\}$) with their matching close-symbols (i.e., $|$). A visibly pushdown automaton can reconstruct the nesting relation by pushing onto the stack on open-tags and popping from it close-tags. This applies to ABTTPMA. ABTTPMA uses stack only for *Block type matching* and Delete operation. Both of these operations push and pop an item to the ABTTPMA's stack in a way that represents the nesting of the input tree. Let S be a set of all transition except those from *Block type matching* and Delete operation. Transitions from the set S push the initial stack symbol Z_0 to the stack. They pop the initial symbol from the stack. If ABTTPMA consists only of transitions from the set S, then ABTTPMA is a finite automaton. Therefore ABTTPMA can be determinized. □

Chapter 4

Implementation

This chapter describes a Java implementation of the proof of concept for the proposed method. As stated, it is implemented in Java — SDK11. Following Java libraries from java.utils were used: ArrayList, Collections, HashSet, Objects. Principles of object-oriented programming—polymorphism, abstraction, and inheritance are applied.

The implemented method (Proof of Concept of the proposed method) is split into two phases, similar to the proposed method. These phases are built the automaton and use the automaton for searching. The automaton is built in a way that states recursively immerse into itself while searching via the automaton is performed. This approach has the advantage that the principle of *Branch and bounds* can be applied to speed up the computation. Recurrent computation approach makes code more transparent, and easy-to-read too.

4.1 Space Optimalisation

Each construction proposed in Section 3.3 is space-intensive as it requires both cloning the base of ABTTPMA and cloning transitions. These clonings happen only because the accumulative counter of edit distance cost is needed. However, if the counter is implemented as a number that cannot overcome a particular value, there is no need to have multiple levels and multiple cloned transitions in the ABTTPMA. Therefore, the counter itself is implemented as an integer, that in every moment, represent the total cost of used edit operations. To proceed from a state to another, To proceed from a state to another, the counter's value must be less or equal to the value of the maximal edit distance. The value of the counter is recursively passed from a state to the next state, as ABTTPMA is searching. By this optimization, we save memory resources because:

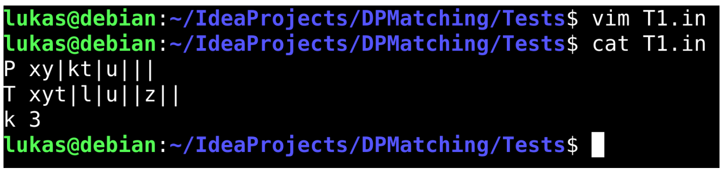
- there are no unreachable states,
- states are not cloned, they can be repeatedly visited during one search phase,
- transitions are not cloned; they can be repeatedly used during one search phase.

4.2 Methods Description

A user can create a new ABTTPMA by calling a constructor `ABTTPMA(String P, int k)` of class `ABTTPMA`. As an input, a **valid** prefix bar notation of tree pattern and maximal edit distance is required. The constructor creates an object of type `ABTTPMA`. This object can call `search(String T)` method. As input, the method requires a **valid** input tree in prefix bar notation. The output of this method is a collection of possible results. Results are also accessible from a public static class `Globals` via it's public method `getResults()`. User can benefit from the overridden `toString()` method for instances of class `Tree`. This method returns a tree rendered as an *ASCII* art.

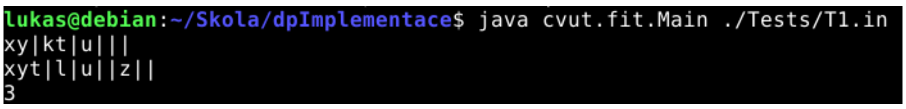
4.3 Implementation Usage

Running our *java* Main.class requires a path to an input file. The file name should have ".in" extension. The file consists of three lines. The first line starts with sequence of two symbols "P ". This sequence is appended by prefix bar notation of a tree pattern P. The second line starts with sequence of two symbols "T ". This sequence is appended by prefix bar notation of an input tree T. The third line starts with sequence of two symbols "k ". This sequence is appended by an integer number that represents maximal edit distance. See Figure 4.1 for an example input file. See Figure 4.2 for example run configuration for running the PoC (Main.class file). The output is saved into a file named `filename`.out. The output file consists of visualization and statics of both trees, edit scripts for each found occurrence, and measured time for build and search phase. See the files in the attached SD card for example datasets. The project source files, run configurations (for IntelliJ Idea project), and test data are also available on Gitlab—`https://gitlab.fit.cvut.cz/rencluka/dpimplementace`.



```
lukas@debian:~/IdeaProjects/DPMatching/Tests$ vim T1.in
lukas@debian:~/IdeaProjects/DPMatching/Tests$ cat T1.in
P xy|kt|u|||
T xyt|l|u||z||
k 3
lukas@debian:~/IdeaProjects/DPMatching/Tests$ 
```

Figure 4.1: Example input file.



```
lukas@debian:~/Skola/dpImplementace$ java cvut.fit.Main ./Tests/T1.in
xy|kt|u|||
xyt|l|u||z||
3
```

Figure 4.2: Example run configuration.

Chapter 5

Testing

For the conducted tests, we used a ThinkPad laptop with i5-7200U CPU @ 2.50GHz with TurboBoost up to 3.1Hz, 16 GB of DDR4 RAM with Debian 10 “Buster”. Presented results are calculated averages from 50 runs of each test.

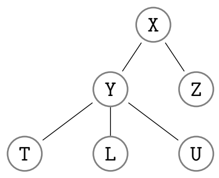
5.1 Validation Testing

Validation tests are conducted to prove the correctness of the implementation. These tests cover all of the possible cases that can occur in input data for Approximate Block Type Matching.

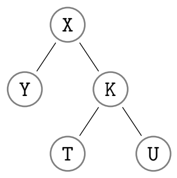
5.1.1 Validation of Edit Operations

To test if all edit opearation work together as intended a special pair of a tree pattern P and an input tree T was created. See Figure 5.1 for this pair. All three edit operations and *Block type matching* are engaged while searching for P in T. This sample pair also shows that there can multiple valid different occurences of P in T where $BlckDist(P, T) = k$. See Figure 5.2. Every test run (both validation and performance tests) was carefully examined if the result is correct. See the files in the attached SD card for complete tests data. Each edit operation is highlighted in a different color.

- Inserted vertices are highlighted in the green color.
- Deleted vertices are highlighted in the red color.
- Relabeled vertices are highlighted in the blue color.



Input tree T.



Tree pattern P.

Figure 5.1: Sample pair of trees P, T for edit operation testing purpose.

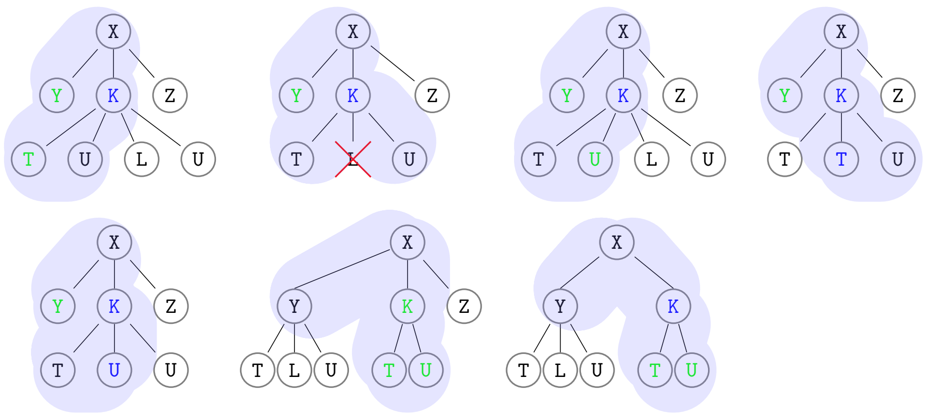


Figure 5.2: All occurences of tree pattern P in input tree T from Figure 5.1 with maximal edit distance $= 3$.

5.1.2 Validation of *Block Type Matching*

There are ten primary samples with different positions of occurrences of tree pattern P in an input tree T. As the samples represent basic structures of trees, any other input tree T can be created by adding subtrees to one of these. Therefore, these samples can test if the implementation of *Block type matching* works correctly. See Figure 5.3 for the prepared samples. The implementation correctly recognized every occurrence of tree pattern P in all ten primary samples presented in Figure 5.3 with block edit distance equal to zero. Therefore, we assume the implementation of *Block type matching* works correctly. See the files in the attached SD card of this thesis for raw test data inputs/outputs.

5.2 Performance Testing

Performance tests measure elapsed time during the search phase for a specific combination of tree pattern P, input tree T, and max edit distance k. Number of vertices in both trees is the same in all tests: $|P| = 7$, $|T| = 140$. Also, the number of occurrences of tree pattern P in the input tree T with edit distance equal to zero is equal to 10. Performance tests experimentally show if elapsed

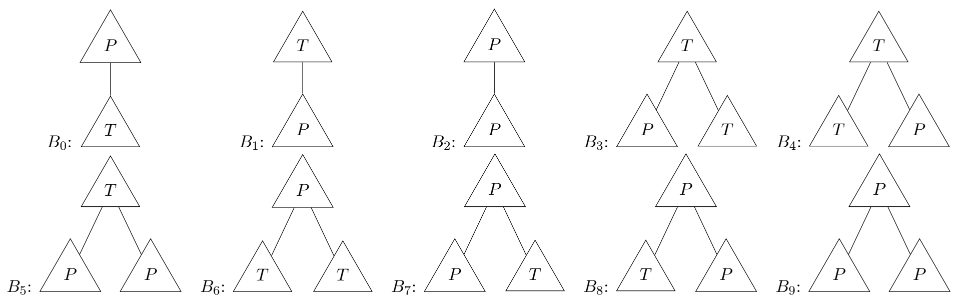


Figure 5.3: Possible positions of occurences of tree pattern P in input tree T.

time for a specific value of max edit distance k is affected by the depth of tree pattern P and input tree T and/or by the inner structure of trees. There are two types and four categories of combinations of P and T. Combinations are split into two types.

- The first type consists of combinations whose tree P has unique vertices.
- The second type consists of combinations whose tree P has repetetive vertices.

Abbreviation **u** stands for "unique". Abbreviation **r** stands for "repetetive".
Categories are considered in respect to depth of P and T.

- Tree P is labeled as shallow if the $depth(P) \leq 3$.
- Tree P is labeled as deep if the $depth(P) \geq 6$.

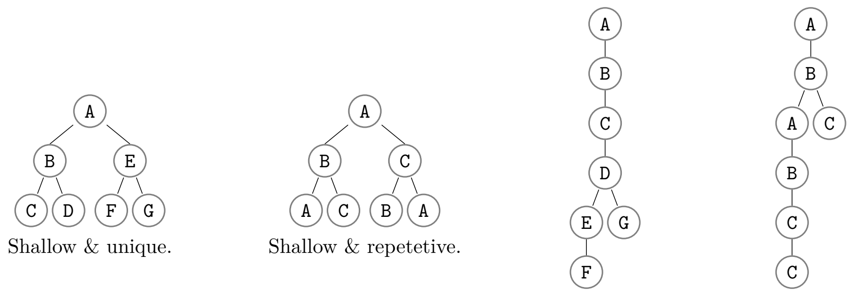
- Tree T is labeled as shallow if the $depth(T) \leq 7$.
- Tree T is labeled as deep if the $depth(T) \geq 30$.

Abbreviation **s** stands for "shallow". Abbreviation **d** stands for "deep". The category name is created by the abbreviated label of the tree P appended by the abbreviated label of the tree T. Categories are: **ss, sd, ds, dd**. See Table 5.1 for transparent visualisation. Tree patterns were created to meet specific criteria of different categories. Figure 5.4 shows tree patterns P used for performance testing. Shallow & unique tree pattern is used with categories **ss** and **sd**. Shallow & repetetive tree pattern P is used with categories **ss** and **sd**. Deep & unique is used with categories **ds** and **dd**. Deep & repetetive is used with categories **ds** and **dd**.

All graphs show the dependency of elapsed time of search phase on input data. As input data to the method, trees of different categories and types were used. Results visualized by blue column used as input data a tree pattern P with unique vertices. Results visualized by red column used as input data a tree pattern P with repetetive vertices. Tests of all combinations of categories and types were conducted with increasing k value—max edit distance.

		shallow T	deep T
Unique vertices	shallow P	t: **u** cat: **ss**	t: **u** cat: **sd**
	deep P	t: **u** cat: **ds**	t: **u** cat: **dd**
Repetetive vertices	shallow P	t: **r** cat: **ss**	t: **r** cat: **sd**
	deep P	t: **r** cat: **ds**	t: **r** cat: **dd**

Table 5.1: Combinations of tree pattern P and input tree T used for testing. There are two types (**t:**) of trees: With unique (**u**) vertices and with repetetive (**r**) vertices. Each type has four categories (**cat:**). The category is an unique combination of shallow (**s**) or deep (**d**) tree pattern P and input tree T. The types and the categories form eight unique combinations of input data for testing purposes.



Shallow & unique. Shallow & repetetive. Deep & unique. Deep & repetetive.

Figure 5.4: Tree patterns P used for performance testing.

Figure 5.5 presents result with k set to $k = 0$. Figure 5.6 presents result with k set to $k = 1$. Figure 5.7 presents result with k set to $k = 2$. Figure 5.8 presents result with k set to $k = 3$.

The results show that the algorithm is sensitive to the structure of the input data. The trees of repetitive type seemed to be more time-consuming to process from all of the possible input data. As labels of the vertices repeat, the algorithm processes more sequences of subtrees that almost match the tree pattern. Therefore, it takes more computation before finding out that such a sequence does not match.

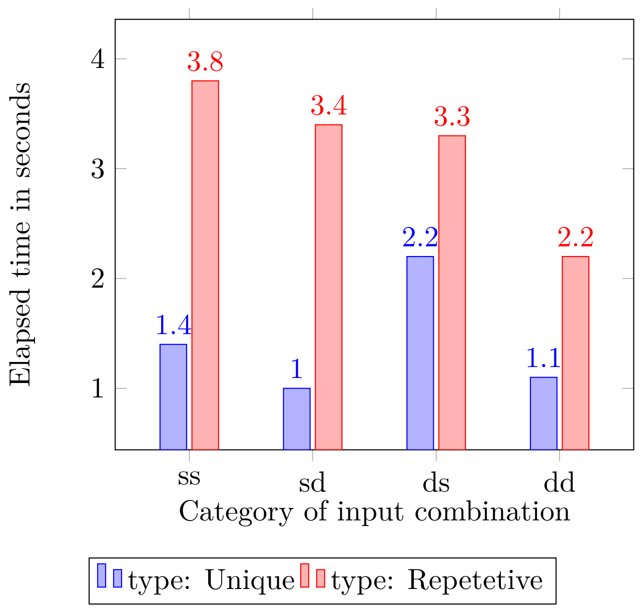


Figure 5.5: Graph of elapsed time with max. edit dist. = 0.

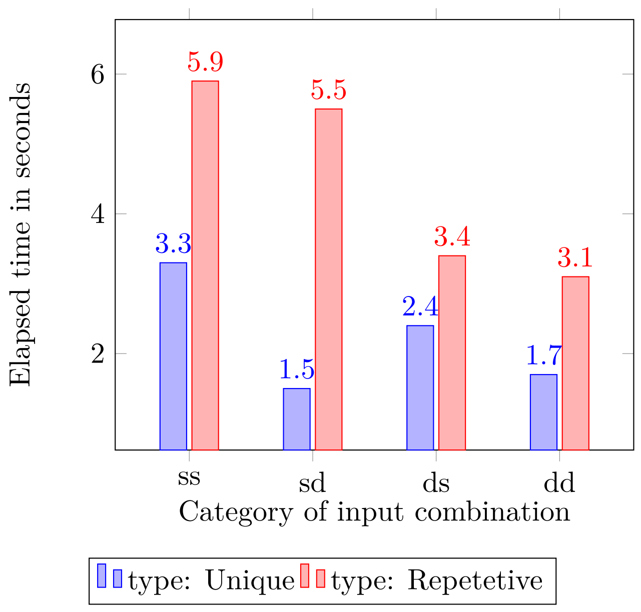


Figure 5.6: Graph of elapsed time with max. edit dist. = 1.

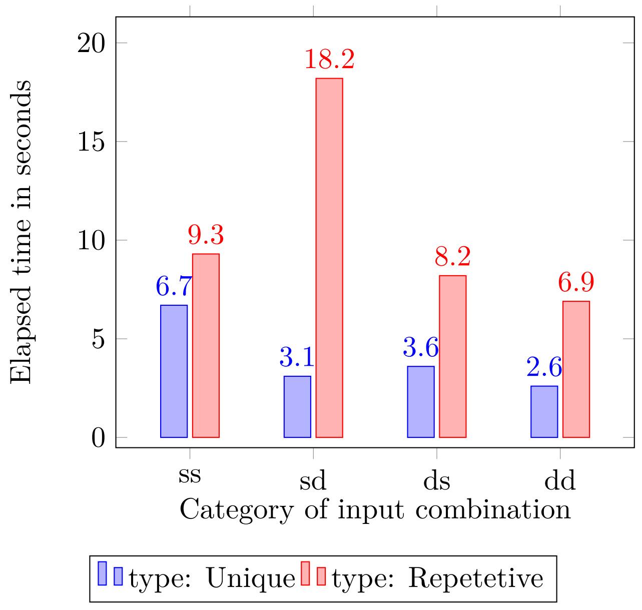


Figure 5.7: Graph of elapsed time with max. edit dist. = 2.

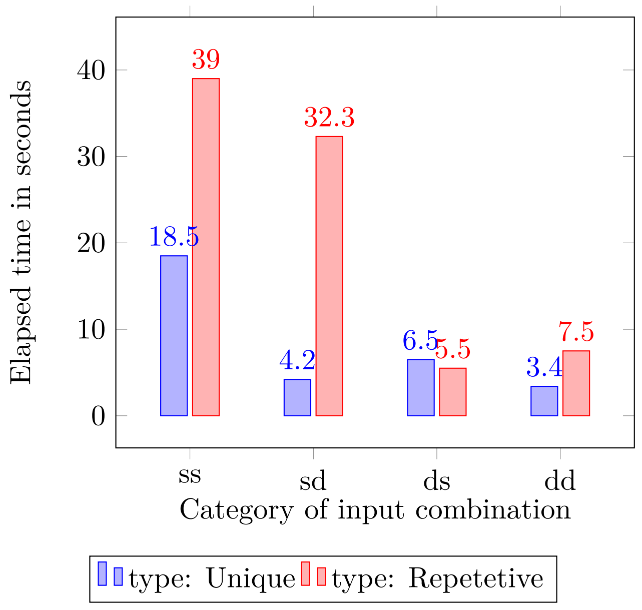


Figure 5.8: Graph of elapsed time with max. edit dist. = 3.

Chapter 6

Conclusion

It is shown that determinizable pushdown automata can be used to solve approximate pattern matching problems. The proposed method creates a pushdown automaton for a given tree pattern P that can find all occurrences of the tree pattern P in an input tree T with maximal edit distance k. Selkow edit operations are used to determine edit distance. The searching phase of the method runs in $\mathcal{O}(k \cdot n \cdot 5^k \cdot 2^{n-k})$, where $n = |prefbar(T)|$. We showed that the resulting automaton belongs to the class of *Visisbly pushdown* automata. These automata are known to be determinizable.

Furthermore, this thesis defines a new type for searching called *Block type matching*. It is implemented into the method. Because of the modular design of the proposed method, the *Block type matching* can be easily substituted with Selkow edit operation Delete.

The implementation of the method involves space optimalization to lower space consumption. Performance tests showed that the method is sensitive to the inner structure of input data.

The thesis LaTeX source code is available at `https://gitlab.fit.cvut.cz/rencluka/DP`. The implementation source codes, test data, and run configurations are available at `https://gitlab.fit.cvut.cz/rencluka/dpimplementace`.

6.1 Goals Fulfillment

The goals were the following:

- Study methods for approximate tree pattern matching and the automata approach for constrained approximate subtree matching introduced in [11].
- Propose a new method for approximate tree pattern matching using the theory of formal languages and automata.

- Implement the proposed method.
- Discuss time and space complexities.
- Perform appropriate testing of the implementation.

We proposed a new method based on the previous studies of approximate tree pattern matching. Chapter 2 sums up conducted research on related work. The newly proposed method combines 1-degree edit distance, pattern matching automata theory, and pushdown automata theory. Additionally, it is extended with our own approach to approximate pattern matching. Implementation of the method is in `Java` programming language. Object-oriented programming principles make the implementation easy to read and debug. The attached code consists of not only the implemented method itself but also of a class for tree visualization and tree parameters viewer. Time and space complexities are discussed in Chapter 3. As the method consists of multiple parts, the time and space complexities are discussed for each part. At the end of the chapter, time complexities are evaluated for the whole method. The testing of the method is presented in Chapter 5. It is split into two parts. Firstly, the function of the method is tested. These tests shall show the correctness of the method. Secondly, performance testing is conducted. These tests are focused on showing if the method behaves differently for various input trees. The results of the conducted tests are evaluated and presented in this thesis. See attached SD card for raw test data.

We state that all goals were fulfilled.

6.2 Future Work

This thesis introduced a new comprehensive method for approximate tree pattern matching. The modular design of its build phase can be easily extended in multiple ways. Firstly, the method can support a new edit operation. As this thesis works with ordered rooted trees, we thought about edit operation called $swap(T_1, T_2)$, where T_1 and T_2 are subtrees of an arbitrary tree T. It would swap the position of two subtrees in a tree. Secondly, the implementation part can be enhanced, as well. This thesis showed that ABTTPMA belongs to *Visibly pushdown* automata class that is known to be determinizable. The implementation can be extended by ABTTPMA determinization. In addition, the experimental test environment can be transformed into an application with GUI.

Bibliography

1. SELKOW, Stanley M. The tree-to-tree editing problem. *Information processing letters*. 1977, vol. 6, no. 6, pp. 184–186.
2. JANOUŠEK, Jan; MELICHAR, Bořivoj. On regular tree languages and deterministic pushdown automata. *Acta Informatica*. 2009, vol. 46, no. 7, pp. 533. Available from DOI: `10.1007/s00236-009-0104-9`.
3. STOKLASA, J; JANOUŠEK, J; MELICHAR, B. Subtree pushdown automata for trees in bar notation, 2010. *London Stringology Days*. 2010.
4. BAXTER, Ira D; YAHIN, Andrew; MOURA, Leonardo; SANT'ANNA, Marcelo; BIER, Lorraine. Clone detection using abstract syntax trees. In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 1998, pp. 368–377. Available from DOI: `10.1109/ICSM.1998.738528`.
5. BROWNE, Cameron B et al. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*. 2012, vol. 4, no. 1, pp. 1–43. Available from DOI: `10.1109/TCIAIG.2012.2186810`.
6. HUDDLESTON, Scott; MEHLHORN, Kurt. A new data structure for representing sorted lists. *Acta informatica*. 1982, vol. 17, no. 2, pp. 157–184. Available from DOI: `10.1007/BF00288968`.
7. BRÜNING, Jens; FORBRIG, Peter. TTMS: A task tree based workflow management system. In: *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2011, pp. 186–200. Available from DOI: `10.1007/978-3-642-21759-3_14`.
8. BURTSCHER, Martin; PINGALI, Keshav. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In: *GPU computing Gems Emerald edition*. Elsevier, 2011, pp. 75–92. Available from DOI: `10.1016/B978-0-12-384988-5.00006-1`.

9. YANG, Wuu. Identifying syntactic differences between two programs. *Software: Practice and Experience*. 1991, vol. 21, no. 7, pp. 739–755. Available from DOI: 10.1002/spe.4380210706.
10. TAI, Kuo-Chung. The tree-to-tree correction problem. *Journal of the ACM (JACM)*. 1979, vol. 26, no. 3, pp. 422–433. Available from DOI: 10.1145/322139.322143.
11. ŠESTÁKOVÁ, Eliška; MELICHAR, Borivoj; JANOUŠEK, Jan. Constrained Approximate Subtree Matching by Finite Automata. In: *Prague Stringology Conference 2018*. 2018, p. 79.
12. HOPCROFT, John E; MOTWANI, Rajeev; ULLMAN, Jeffrey D. Introduction to automata theory, languages, and computation. *Acm Sigact News*. 2001, vol. 32, no. 1, pp. 60–65. Available from DOI: 10.1145/568438.568455.
13. CORMEN, Thomas H; LEISERSON, Charles E; RIVEST, Ronald L; STEIN, Clifford. *Introduction to algorithms*. MIT press, 2009. ISBN 978-0-262-03384-8.
14. KNUTH, Donald Ervin. *The art of computer programming*. Pearson Education, 1997. ISBN 0-201-03801-3.
15. VAN TANG, Nguyen. A tighter bound for the determinization of visibly pushdown automata. *arXiv preprint arXiv:0911.3275*. 2009. Available from DOI: 10.4204/EPTCS.10.5.
16. OKHOTIN, Alexander; SALOMAA, Kai. Complexity of input-driven pushdown automata. *ACM SIGACT News*. 2014, vol. 45, no. 2, pp. 47–67. Available from DOI: 10.1145/2636805.2636821.
17. ALUR, Rajeev; MADHUSUDAN, Parthasarathy. Visibly pushdown languages. In: *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. 2004, pp. 202–211. Available from DOI: 10.1145/1007352.1007390.
18. NOWOTKA, Dirk; SRBA, Jiří. Height-deterministic pushdown automata. In: *International Symposium on Mathematical Foundations of Computer Science*. 2007, pp. 125–134. Available from DOI: 10.1007/978-3-540-74456-6_13.
19. MEHLHORN, Kurt. Pebbling mountain ranges and its application to DCFL-recognition. In: *International Colloquium on Automata, Languages, and Programming*. 1980, pp. 422–435. Available from DOI: 10.1007/3-540-10003-2_89.
20. BRAUNMÜHL, Burchard von; VERBEEK, Rutger. Input-driven languages are recognized in log n space. In: *International Conference on Fundamentals of Computation Theory*. 1983, pp. 40–51. Available from DOI: 10.1007/3-540-12689-9_92.

21. DYMOND, Patrick W. Input-driven languages are in log n depth. *Information processing letters*. 1988, vol. 26, no. 5, pp. 247–250. Available from DOI: `10.1016/0020-0190(88)90148-2`.
22. HOLZER, Markus; KUTRIB, Martin; MALCHER, Andreas; WENDLANDT, Matthias. Input-Driven Double-Head Pushdown Automata. *EPTCS 252*, pp. 128. Available from DOI: `10.4204/EPTCS.252.14`.
23. ALUR, Rajeev; MADHUSUDAN, Parthasarathy. Visibly pushdown languages. In: *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. 2004, pp. 202–211. Available from DOI: `10.1145/1007352.1007390`.
24. NOWOTKA, Dirk; SRBA, Jiří. Height-deterministic pushdown automata. In: *International Symposium on Mathematical Foundations of Computer Science*. 2007, pp. 125–134. Available from DOI: `10.1007/978-3-540-74456-6_13`.
25. POLÁCH, Radomír; TRÁVNÍČEK, Jan; JANOUŠEK, Jan; MELICHAR, Bořivoj. Efficient determinization of visibly and height-deterministic pushdown automata. *Computer Languages, Systems & Structures*. 2016, vol. 46, pp. 91–105. Available from DOI: `10.1016/j.cl.2016.07.005`.
26. OHST, Dirk; WELLE, Michael; KELTER, Udo. Differences between versions of UML diagrams. In: *ACM SIGSOFT Software Engineering Notes*. 2003, vol. 28, pp. 227–236. No. 5. Available from DOI: `10.1145/940071.940102`.
27. CHAWATHE, Sudarshan S; RAJARAMAN, Anand; GARCIA-MOLINA, Hector; WIDOM, Jennifer. Change detection in hierarchically structured information. *Acm Sigmod Record*. 1996, vol. 25, no. 2, pp. 493–504. Available from DOI: `10.1145/235968.233366`.
28. CHAWATHE, Sudarshan S; GARCIA-MOLINA, Hector. Meaningful change detection in structured data. *ACM SIGMOD Record*. 1997, vol. 26, no. 2, pp. 26–37. Available from DOI: `10.1145/253262.253266`.
29. BARNARD, David T; CLARKE, Gwen; DUNCAN, Nicolas. Tree-to-tree correction for document trees. 1995. Available from DOI: `10.1.1.29.5248`.
30. ZHANG, Kaizhong; SHASHA, Dennis. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*. 1989, vol. 18, no. 6, pp. 1245–1262. Available from DOI: `10.1137/0218082`.
31. OOMMEN, John B. *Method of comparing the closeness of a target tree to other trees using noisy sub-sequence tree processing*. Google Patents, 2007. US Patent 7,287,026.

32. MARTIN-VIDE, Carlos; MATEESCU, Alexandru; MITRANA, Victor. Parallel finite automata systems communicating by states. *International Journal of Foundations of Computer Science*. 2002, vol. 13, no. 05, pp. 733–749. Available from DOI: `10.1142/S0129054102001424`.

33. MELICHAR, Bořivoj. Arbology: Trees and pushdown automata. In: *International Conference on Language and Automata Theory and Applications*. 2010, pp. 32–49. ISBN 978-3-642-13088-5. Available from DOI: `10.1007/978-3-642-13089-2_3`.

34. MELICHAR, Bořivoj. Approximate string matching by finite automata. In: *International Conference on Computer Analysis of Images and Patterns*. 1995, pp. 342–349. Available from DOI: `10.1007/3-540-60268-2_315`.

Appendix **A**

Acronyms

ABTTPMA Approximate block type tree pattern matching automaton

PoC Proof of concept

SDK Software development kit

ASCII American standard code for information interchange

XML Extensible markup language

HTML Hypertext markup language

CAD Computer-aided design

UML Unified Modeling Language

Apx Approximate

CPU Central processing unit

RAM Random-access memory

GUI Graphical user interface

Appendix **B**

Contents of Enclosed SD Card

```
readme.txt ........................ the file with SD card contents description
├── thesis ............................................... the directory of thesis
│   ├── thesis.pdf ................................. thesis text in PDF format
│   └── src .................... the directory of LaTeX source codes of the thesis
└── implementation ................................... the thesis text directory
    ├── src ............... the directory with source codes of implementation
    └── tests ............................ the directory with tests input data.
```