FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	A Survey on Tree Indexing Problem
Student:	Karolina Hrnčiříková
Supervisor:	Ing. Eliška Šestáková
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2021/2022

Instructions

Make a survey on tree indexing problems and definitions used in literature.
Upon agreement with the supervisor focus on one problem (i.e., variant of tree indexing) and make a thorough research on existing algorithms solving this problem.
Moreover, upon agreement with the supervisor select two of the algorithms and implement them.
Choose/prepare data with various characteristics (such as deep/shallow trees), and perform experiment evaluation of time and memory consumption.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 19 January 2021 in Prague.

Bachelor’s thesis

A SURVEY ON TREE INDEXING PROBLEM

Karolina Hrnčiříková

Faculty of Information Technology
Department of Theoretical Computer Science
Supervisor: Ing. Eliška Šestáková
May 9, 2022

Contents

List of Figures

List of Algorithms

I would like to express my gratitude and thanks to my thesis supervisor Eliška Šestáková. Her experience and insight into the topic, as well as her focus on detail, were essential to the work. She had unwavering patience with me and provided me with know-how far beyond the topic of tree indexing. Special thanks go to Tomáš Pecka, who greatly helped me to improve the implementation part of the thesis. Great thanks go to my mom and dad for their unconditional and enthusiastic support, to my boyfriend for being the best rubber duck a programmer could ask for, and to my friends and fellow students for making the school a little easier and a whole lot more fun.

Declaration

In Prague on May 9, 2022 ....

Abstract

A tree indexing problem is a special kind of tree pattern matching that preprocesses a given input tree to optimize locating one type of query within it. There are many variants of tree indexing problems, and to the best of our knowledge, there does not exist an overview of them. Therefore, we survey existing tree indexing problems and provide categories that help define them. The categories are the following: the type of input tree, the type of queries, what is considered a match, and the type of required answers. One tree indexing problem that we defined using these categories is the subtree rejection problem. This problem focuses on whether a given subtree is present in an input tree rather than where. We discuss two existing solutions to the subtree rejection problem. The solutions are subtree pushdown automata and subtree oracle pushdown automata. Moreover, we also propose an index based on a finite automaton, called the subtree finite automaton. Given an input tree and a query pattern, the query time of all three methods depends only on the size of the query pattern and not on the size of the preprocessed input tree. To assess the efficiency of the solutions in practice, we compare their performance on ranked ordered trees with heights of up to 15. In these experiments, subtree finite automata perform the best with respect to build and query time.

Keywords tree indexing, index, pushdown automaton, finite automaton, oracle, subtree, pattern matching

Abstrakt

Stromové indexování je problém vyhledávání určitého typu dotazů v předzpracovaných datech, které mají tvar stromové struktury. Tento problém existuje v mnoha variantách. Jelikož jsme nenašli shrnutí, které by popisovalo různé varianty, v této práci jsme vypracovali jejich rešerši. V rámci ní jsme definovali čtyři kategorie, pomocí kterých lze varianty stromového indexování klasifikovat. Tyto kategorie jsou: typ indexovaného stromu, typ dotazů, co se počítá jako shoda a typ požadovaných odpovědí. Dále popisujeme tři řešení k tzv. *subtree rejection* problému, který se zabývá tím, zda se někde ve vstupním stromě nalézá daný podstrom. Řešení, která probíráme, jsou *subtree pushdown automata*, *subtree oracle pushdown automata* a nový algoritmus *subtree finite automata*. Rychlost odpovídání na dotazy se u všech tří řešení odvíjí od velikosti dotazu a nezáleží na velikosti předzpracovaného stromu. Abychom porovnali efektivitu těchto řešení v praxi, implementovali jsme je a otestovali na ohodnocených seřazených stromech s hloubkou maximálně 15. Nová indexovací technika subtree finite automata má nejlepší výsledky co se týče rychlosti postavení indexu a rychlosti odpovězení dotazu.

Klíčová slova stromové indexování, index, zásobníkový automat, konečný automat, orákulum, podstrom, vyhledávání vzorků

Abbreviations

FA	Finite Automaton
PDA	Pushdown Automaton
XML	Extensible Markup Language

Chapter 1
Introduction

When working with hierarchical data, it is often convenient to use a tree structure to represent them. Searching the data for a specific class of patterns is called the *tree indexing problem*. The aim of tree indexing problem solutions is to preprocess a given input tree so that we can find all occurrences of query patterns within the tree in the shortest amount of time possible. Many areas rely on such tree indexing algorithms. These areas are, for example, disciplines in bioinformatics such as phylogeny or work with DNA and RNA structures [1], or in computer science, compiler code optimizations [2], file systems, and XML query languages.

Due to the varied applications of tree indexes, there are many tree indexing problems, each with nuances that match the intended use. The aim of this thesis is to explore the different variants of tree indexing problems and categorize them. We describe four properties that can be used to define a tree indexing problem. The properties are:

- the type of trees we index,
- the type of patterns we are looking for,
- what we consider to be a match, and
- the type of answers we require.

Tree indexing is also a problem that can be quite easily expanded to other tasks. These tasks can be considered more complex since the focus is not on answering a simple occurrence question but rather on drawing some conclusions based on the structure of the data. For example, many publications focus on finding a common pattern in a vast dataset. The problem is often called *searching for tree repetitions* [2, 3], or *tree mining problem* [1].

1.1 Aim of the thesis

The aim of this thesis is to research tree indexing problems. The study consists of three main parts. First, we aim to survey tree indexing problems and definitions used in the literature. Second, our aim is to choose and study one of the problems described in the first part of this thesis in more depth. We mean to identify possible approaches to the problem and compare and contrast some of its solutions. Our final aim is to implement two solutions to the subtree rejection problem and to compare their efficiency. For that, we must prepare data and experiments that measure time and memory consumption.

1.2 Structure of the thesis

The rest of the thesis is organized as follows. The second chapter lays the foundation for the thesis by defining the necessary terms used in this work. Namely, we present basic definitions from graph theory such as graph, tree, and forest. Then we define concepts from automata theory, specifically pushdown and finite automata. The third chapter shows the results of our research on tree indexing problems. The fourth chapter explains and compares three solutions to the subtree rejection problem, specifically the subtree pushdown automata, the subtree oracle pushdown automata, and the subtree finite automata. The fifth chapter presents the implementations of the three solutions. The sixth chapter compares the performance of the three implementations. The last chapter summarizes the most important results of the thesis.

Chapter 2
Preliminaries

This chapter defines the terms and notation used throughout the thesis. It can be skipped and referred back to it as necessary. All the definitions are analogous to how they are defined in graph theory and formal languages. Several works were used as a reference point for this chapter [4, 5, 6, 7, 8, 9].

2.1 Alphabet, string, language

An *alphabet* is a finite non-empty set of symbols denoted by A. An alphabet can be *ranked*, which means that each symbol of the alphabet has an associated *arity* with it. The arity of a symbol is a non-negative integer denoted by $\text{arity}(a)$.

The symbols of an alphabet are the building blocks of strings. Formally, a *string* is a finite sequence of symbols over a given alphabet A. The set of all strings over A is denoted by A^*, and the set of all non-empty strings over A is denoted by A^+. The *length* of a string is a non-negative integer that represents the number of symbols in a string. For a string x, the length is denoted by $|x|$. Furthermore, we define a special string, *empty string*, of zero length, denoted by ε. In the thesis, we use the letters x, y, and z for strings, and the letters a, b, and c for symbols of an alphabet.

A *language* L over an alphabet A is a set of strings over A, formally $L \subseteq A^*$. The *size* of a language is equal to the number of strings in L. A language is *finite* when it contains a finite number of strings.

Throughout this thesis, we refer to various parts of a string. Let us have a non-empty string $x = a_1a_2a_3 \dots a_n$ of length n. A *prefix* of x is any string $y = a_1a_2a_3 \dots a_m$ where $1 \leq m \leq n$. A *suffix* of x is any string $y = a_ia_{i+1}a_{i+2} \dots a_n$ where $1 \leq i \leq n$. A *factor (substring)* of x is any string where $y = a_ia_{i+1}a_{i+2} \dots a_j$ where $1 \leq i \leq j \leq n$. A *subsequence* of x is any non-empty string that can be obtained by deleting zero or more symbols from x.

▶ **Example 2.1** (Alphabet, string, language)**.** Let us have an alphabet $A = \{a, b, c\}$ and a string $x = cbbab$. Then, $|x| = 5$. The set of all prefixes is $\{c, cb, cbb, cbba, cbbab\}$. The set of all suffixes is $\{b, ab, bab, bbab, cbbab\}$. The set of all factors of x includes all its prefixes, suffixes, and more, for example: b, a, c, cb, ba, ab, bba, cbb, or bab. The set of all subsequences includes all factors and more, for example: cb, ab, ca, cab, $cbbb$, or bbb.

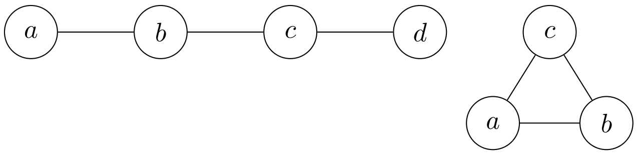


Figure 2.1 On the left is a graph containing a path of length 3 and on the right is a graph containing a cycle of size 3.

2.2 Graph, tree, forest

In this section, we formally define the notion of a tree. We achieve that by using the concepts formulated in graph theory. First, we review the notion of a graph. Second, we state the definition of a tree and its properties. Finally, we review the different types of trees.

A *graph* is a pair (V, E), where V is a set of *vertices* and E is a set of unordered pairs of vertices called *edges*; if p and q are vertices, then $[p, q]$ denotes that there is an edge between p and q. There are two particular parts of a graph that we need to define. A *path in a graph* is a sequence of distinct vertices $(v_0, v_1, v_2 \dots v_m)$, where $m \geq 1$. The path has m edges so that for every $i \in \{1, 2 \dots m\}$ there is an edge $[v_{i-1}, v_i]$. The *length* of a path is defined by the number of edges on the path. A *cycle in a graph* is a sequence of vertices $(v_0, v_1, v_2 \dots v_m)$, where $m \geq 3$ and all vertices are distinct except v_0 and v_m, which are equal. The cycle has m edges, so for every $i \in \{1, 2 \dots m\}$ there is an edge $[v_{i-1}, v_i]$. The *size* of a cycle is the number of vertices (edges) in the cycle. For example, Figure 2.1 shows a graph containing a path of length 3 and a graph containing a cycle of size 3.

A graph can have different characteristics. Let us have a graph $G = (V, E)$. G is *connected* when there is a path between every two vertices in V. G is called *labeled* when it has a mapping from the vertices of G to symbols of an alphabet. Every vertex $v \in V$ has a property called a degree. The *degree* of a vertex v is the number of edges $[v, w] \in E$, where $w \in V$.

The following paragraphs formally define trees in graph theory. A *tree* is a connected graph without cycles, sometimes also called a *free tree*. A *forest* is a graph without cycles (a set of trees). By convention, we call the vertices of a tree *nodes*. In the following, we define some of the different types of trees.

▶ **Definition 2.2** (Rooted tree)**.** Let us have a tree (V, E). A *rooted* tree has one special node $r \in V$ called the *root* of the tree.

When a diagram depicts a rooted tree, its root is typically represented by a node at the top of a tree. For example, in Figure 2.2, we can see a tree with its root labeled by a.

Additionally, we describe various relationships between nodes in rooted trees. Let (V, E) be a rooted tree with nodes $v, w \in V$. Node w is a *child* of v and v is the *parent* of w when there is an edge $[v, w]$ between them and the path from the root node to v is shorter than that from the root to w. The nodes v and w are *siblings* when they have the same parent. The node w is a *descendant* of v and v is an *ancestor* of w when there exists a path from v to w on a path from the root node to w. Moreover, we define a specific type of node called a leaf. A *leaf* is a node without children.

A tree has certain attributes that we measure. A *height* of a tree is the number of edges on the longest path from the root node to a leaf node. A *size* of a tree is the size of the set V.

▶ **Definition 2.3** (Labeled tree)**.** A tree is *labeled* when the underlying graph is labeled.

▶ **Definition 2.4** (Ranked tree)**.** Let us have a ranked alphabet A and a tree $T = (V, E)$. A *ranked tree* T is a labeled rooted tree, where the labeling is over A; and the number of children of every node $v \in V$ with label $a \in A$ is defined by $\text{arity}(a)$.

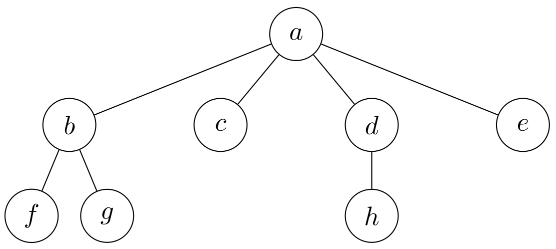


Figure 2.2 An example of a rooted labeled tree.

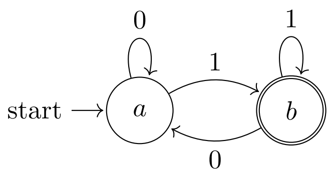


Figure 2.3 A deterministic FA $M = (\{a, b\}, \{1, 0\}, \delta, a, \{b\})$, where δ transitions are $\delta(a, 0) = a$, $\delta(a, 1) = b$, $\delta(b, 0) = a$, and $\delta(b, 1) = b$. M accepts all odd binary numbers.

▶ **Definition 2.5** (Ordered tree)**.** Let us have a rooted tree $T = (V, E)$. T is *ordered* when for each node $v \in V$ its children are ordered.

▶ **Example 2.6** (Height and size of a tree, degree of a node)**.** Let us have Figure 2.2 as an example of a rooted labeled tree. The height of the tree is 2 and its size is 8. The root of the tree is the node labeled a. The node a has a degree equal to 4 and b has a degree equal to 3. The nodes f and g are siblings. The node b is the parent of g and g is a child of b. The node f is a descendant of a and a is an ancestor of f.

2.3 Finite and pushdown automata

One approach to indexing trees, which we discuss in Chapter 3, uses automata as its model of computation. In this section, we define two types of automata. First, we define a finite automaton (FA) and then a pushdown automaton (PDA). Both of these are shown in their deterministic and nondeterministic versions.

A *deterministic finite automaton* is a quintuple (Q, A, δ, q_0, F), where Q is a finite set of states, A is an alphabet, δ is a mapping from $Q \times A$ to Q, elements of which are called *transitions*, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of final states. In a *nondeterministic finite automaton*, δ is a mapping from $Q \times A$ into the power set of Q.

We use configurations to describe the steps of computation of both nondeterministic and deterministic finite automata. Let there be a finite automaton $M = (Q, A, \delta, q_0, F)$. A *configuration* of M is an ordered pair $(q, x) \in Q \times A^*$. Let there be a relation over the configurations $\vdash_M \subset (Q \times A^*) \times (Q \times A^*)$. An element of the relation $\vdash_M$ is called *move.* If $p \in \delta(q, a)$ or $\delta(q, a) = p$ (in the case of a deterministic FA), then $(q, ax) \vdash_M (p, x)$, where $a \in A$, $x \in A^*$, and $p, q \in Q$. Furthermore, $\vdash_M^+$ means that there is at least one move between the two configurations, and $\vdash_M^*$ means that there is any number of moves. A finite automaton *accepts* a string x when $(q_0, x) \vdash^* (q, \varepsilon)$, where $q \in F$.

The *language* accepted by a finite automaton is the set of strings accepted by the automaton. Two finite automata are *equivalent* if they accept the same language. We can convert every nondeterministic FA into an equivalent deterministic FA. An example of a deterministic finite automaton is shown in Figure 2.3.

A *nondeterministic pushdown automaton* is a 7-tuple $(Q, A, G, \delta, q_0, Z_0, F)$, where Q is a finite set of states, A is an alphabet, G is a pushdown store alphabet, δ is a mapping from a finite subset $Q \times (A \cup \{\varepsilon\}) \times G^*$ into a set of finite subsets $Q \times G^*$ called transitions, $q_0 \in Q$ is an initial state, $Z_0 \in G$ is an initial pushdown store symbol, $F \subseteq Q$ is a set of final states. A *deterministic pushdown automaton* means that the following rules hold for $\forall q \in Q, \forall a \in (A \cup \{\varepsilon\}), \forall \gamma, \beta, \alpha \in G^*$:

- $|\delta(q, a, \gamma)| \leq 1$. Informally, for each starting point of a transition, there is at most one possible outcome.
- If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$, then α is not a prefix of β and vice versa. Informally, when there are two transitions from the same state on the same symbol on the input, then they have to differ in what they expect on the pushdown store.
- If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, \varepsilon, \beta) \neq \emptyset$, then α is not a prefix of β and vice versa. Informally, when there are two transitions from the same state, and at least one does not accept anything from the input, they have to be completely different in what they expect on the pushdown store.

To describe the computation steps of a PDA, we also use configurations. Let there be a PDA $M = (Q, A, G, \delta, q_0, Z_0 F)$. A *configuration* of M is an ordered triple $(q, x, \gamma) \in Q \times A^* \times G^*$. Let there be a relation over the configurations $\vdash_M \subset (Q \times A^* \times G^*) \times (Q \times A^* \times G^*)$. If $(p, \beta) \in \delta(q, a, \alpha)$, then $(q, ax, \alpha\gamma) \vdash_M (p, x, \beta\gamma)$, where $a \in A$, $x \in A^*$, $\alpha, \beta, \gamma \in G^*$, and $p, q \in Q$. Furthermore, same as with finite automata, $\vdash_M^+$ means that there is at least one move between the two configurations and $\vdash_M^*$ means that there is any number of moves.

The *language* accepted by a pushdown automaton M is the set of strings accepted by the automaton, denoted by $L(M)$. Similar to finite automata, two pushdown automata are *equivalent* when they accept the same language.

In contrast to finite automata, pushdown automata have two ways of accepting a string. In this thesis, we only use pushdown automata that accept a language by having an empty pushdown store. That is, a PDA $M = (Q, A, G, \delta, q_0, Z_0, F)$ accepts a language when:

$$L(M) = \{x : x \in A^*, \exists q \in Q, (q_0, x, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}.$$

By convention, to signify that a PDA accepts by an empty pushdown store, we denote the set of final states by the empty set $\emptyset$.

In contrast to finite automata, not every nondeterministic PDA has an equivalent deterministic PDA that accepts the same language. Therefore, we recognize the types of pushdown automata that have the ability, which includes input-driven pushdown automata. An *input-driven* PDA determines each pushdown operation only by the symbol on the input [4]. Furthermore, a PDA is *acyclic* if it does not contain transitions that would allow visiting one state, accepting some symbols on the input, and revisiting the same state [10].

In diagrams, a transition labeled $a, \gamma/\alpha$ from a state q to a state p means that $(p, \alpha) \in \delta(q, a, \gamma)$, where $a \in A \cup \{\varepsilon\}$ is what we accept on the input, $\gamma \in G^*$ is the content at the top of the pushdown store that is erased, and $\beta \in G^*$ is what is written at the top of the pushdown store after the move. In Figure 2.4, we see an example of a deterministic PDA and its diagram.

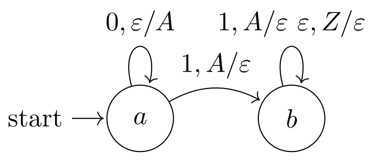


Figure 2.4 A deterministic PDA $M = (\{a, b\}, \{0, 1\}, \{A, Z\}, \delta, a, Z, \emptyset)$. M accepts a string by an empty pushdown store and accepts a language, where every string is in the form 0^n1^n.

Chapter 3

Survey on tree indexing problems

Indexing is a special kind of pattern matching that preprocesses given data to optimize locating one type of query within them. We speak of the *tree indexing problem* when the data are in the form of a hierarchical structure that a tree or a forest can represent. The query type can be anything that, when successful, matches onto a subgraph in the tree data, occasionally accompanied by other information. The given data are typically preprocessed only once to create their index. Afterward, the indexing structure can be repeatedly queried to find occurrences of a specific class of query patterns in the subject data. The general scheme of tree indexing can be seen in Figure 3.1.

▶ **Problem 3.1** (Tree indexing problem)**.** Let there be a tree T. The *tree indexing problem* is to build an indexing structure for T that can answer a specific type of query patterns.

However, the problem described above is a very vague definition of tree indexing. Since trees are used in various ways and disciplines, there are many tree indexing problems. We have found four key properties that together define a tree indexing problem. In this chapter, we look at the different types of the problem and at each of the different properties and review the categories within them.

The first property defines the type of tree structure that we index. The second one states the different types of query. The third property reviews the different conditions under which an index accepts a query. Finally, the fourth property defines the possible output of an index.

Furthermore, in the last two sections of this chapter, we describe the approaches and objectives of the solution algorithms. In those sections, we overview the different approaches to solving tree indexing problems and the memory requirements of the indexing algorithms.

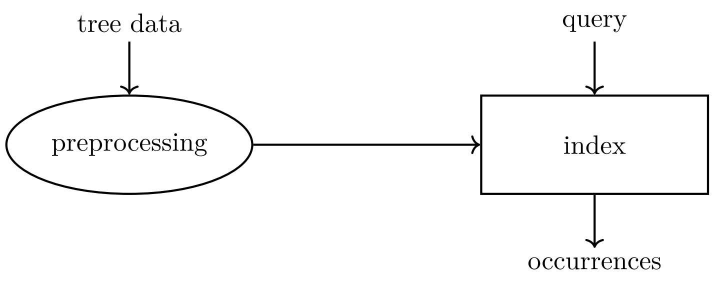


Figure 3.1 A diagram illustrating the tree indexing problem: an input tree is preprocessed so that its index is created. The index can be given a query pattern to which it finds all its occurrences in the input tree.

3.1 Tree type

In this section, we focus on the different types of input trees for the tree indexing problem with a brief overview of their solutions. The input tree type is the first property that helps to define an indexing problem. The type of tree for which a solution can create an indexing structure is strictly defined, as solutions to one type of tree can be incompatible with other types of trees. However, solutions to one type of tree can sometimes be modified to index different types. This section provides an overview of the most common general tree types that are indexed. Additionally, we go over two specific tree structures. However, there are many more specific use cases that define the tree structure more strictly that we do not discuss, such as databases, file systems, and JSON indexing.

Ordered trees are one of the most commonly indexed trees. Most of the solutions specifically for ordered trees that we have found take the arbology approach to indexing [11, 10, 12, 13, 14] with the exception of the approximate indexing solution [15] and the compression and indexing algorithms [16]. Since arbology-based approaches use prefix notations of trees, solutions based on arbology algorithms do not have a simple modification that would make them compatible with unordered trees. Queries in their prefix notation are substrings (or substrings with gaps) of the prefix notation of an input tree. These substrings are matched with the prefix notation. This means that we cannot change the order of subtrees' nodes and, by extension, of the symbols of the substrings because the subtree's prefix notation would not be otherwise present in the prefix notation of the input tree.

▶ **Problem 3.2** (Ordered tree indexing problem)**.** Let there be an ordered tree T. The *ordered tree indexing problem* is to build an indexing structure for T that can answer a specific type of query pattern.

We found two solutions in addition to arbology algorithms that focus on labeled trees. Cohen [15] presented an algorithm for approximate indexing of subtrees for ordered labeled trees, and Ferragina, Luccio, Manzini, and Muthukrishnan [16] for indexing as well as compressing labeled trees. Additionally, since ranked trees are a special subtype of labeled trees, both of these above-mentioned solutions can be used for ranked trees as well. However, there are many more solutions that focus specifically on ranked trees.

All algorithms that index specifically ranked trees that we found can be modified to index unranked trees [17, 12, 14]. The solutions are based on the arbology approach and work with the prefix notation of trees upon which a PDA is created. Although, the presented algorithms work with the prefix notation (see Definition 3.10) a simple extension of the algorithms could accommodate the prefix bar notation (see Definition 3.11) used for unranked trees. The problems we can solve using these algorithms are indexing for subtrees [12, 14] and (nonlinear) tree patterns [17].

We have found only one solution presented by Chi, Yang, and Muntz [1] that focuses on indexing free trees during our research. The solution is based on rewriting an input free tree into a canonical representation. The canonical representation is a sort of rooted tree that is the same for all isomorphic free trees. It is obtained by deleting all leaves in waves until there are either one or two nodes left. Those nodes are treated as a root, and their children are the lastly deleted leaves, and so forth. A notable application of indexing free trees is in multicast trees in computer networks [1].

3.1.1 XML

Extensible Markup Language (XML) is the state-of-the-art format for sharing and storing data. For XML's widespread use, we chose it to demonstrate that even though XML documents can be seen as labeled ordered trees, the specific requirements posed on its use caused that there are many indexing structures explicitly designed for XML data.

In many cases, it is best to work with data in the format in which they already are rather than converting them once more. To be able to query a document, that is, find substructures in it, many query languages for XML have been created, such as XML Path Language, XML Pointer Language, XML Query, and many more that enable expressing complex queries [18].

Given the similar nature of querying XML languages and indexing, the XML query languages are built on indexing algorithms. There has been extensive research on XML indexing approaches [19, 6, 20]. The approaches differ not only in the query shape and the bases of the query but also in the way in which the XML data themselves are interpreted. One of those categories utilizes the fact that XML documents can naturally form a tree structure. From that point of view, the tree indexing algorithms prove to be helpful.

XML document is semistructured and forms labeled ordered trees. The content is enclosed in various tags. These tags are not predefined but rather created by users, which ensures that the data within are self-describing. Tags are typically the inner nodes of a tree. The values within are the leaves of the tree. Every correctly set up XML document has an element that can be considered the root of a tree [21].

Despite the similarity between XML documents and trees, XML-specific tree indexing faces a few complications. First, the XML format and its languages are flexible, making it easy for us to bend the document shape into a form that best fits our purposes, but it makes it harder to index. Second, some query languages allow for both content-based and structure-based queries, making the search more accurate [20, 18, 22]. Third, an XML document intended for querying is, in many cases, going to change throughout the time we use it. Depending on the frequency of changes, it may be beneficial to allow the index structure to be updated rather than building a new one [18, 23]. Generally, update operations should allow for inserting, deleting, and updating data [18]. All these capabilities are useful to users but make the task of indexing more complicated, as the number of potential queries is often exponential to the size of the document [24] (compared to subtree indexing, where the number of potentially "accepted" queries is linear). In fact, there are $O(2.62^n)$ potential queries present for a linear tree (tree in the shape of a path) when only the child axis and descendant-or-self axis specifications are allowed, illustrating the flexibility that the query languages must accommodate [25].

Therefore, a special branch of tree indexing developed, focusing solely on XML documents. Unfortunately, many of the novelty indexing techniques are not used in practice [19, 26]. *XML Document Indexes: A Classification* offers an overview of many possible techniques. In this chapter, we overview merely those algorithms including tree indexing.

According to certain publications [27, 28, 18], it is fairly standard that XML indexes combine the results of individual simple queries into more complex queries, such as branching queries. We find that trend to be true [29, 30, 31]. It seems that most querying languages take as the basic building block a simple path query. When the query is not as simple, the language divides the problem into smaller sub-problems and joins the results. The process of joining the sub-answers is costly. Therefore, to solve the issue in a reasonable time, there are algorithms for joining path query answers based on the structural index and numbering schemes [28]. However, there are authors who find these techniques insufficient. Some publications follow the notion that it is best to avoid joining simpler answers into a complex query answer altogether. They achieve this by directly indexing the branching query [27, 28, 18]. This is often done using a sequence-based approach that transforms both the document and the query into a sequence [29, 19]. These sequences are then pattern-matched, as is similarly done in arbology.

Another school of thought went a different way to solve the problem of joining queries by creating an index that remembers frequent queries [18]. This means that the index structure has to keep track of the queries, which requires some overhead for maintenance in exchange for high-speed searching for a typical query.

Publications presenting new XML indexing methods often focus on a small category of queries. These algorithms can be tied together through an XML query language, creating a complex index with suitable properties for a specific use.

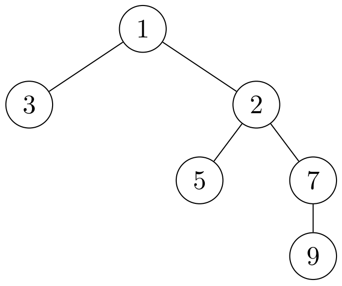


Figure 3.2 Cartesian tree for the string 315279.

3.1.2 Cartesian index

In this thesis, we study the indexing of tree-shaped data. However, there is a noteworthy problem that shows us the variability of tree indexing. The Cartesian index presents a tree indexing problem where, although tree indexing is present, the initial data are not in the form of a tree.

When we have any time-bound numerical data and are more interested in the patterns rather than the exact values, such as stock market and weather patterns, Cartesian trees are a good structure to store the data. In order to manipulate, retrieve, and acquire information, several algorithms on Cartesian trees were created [32, 33]. Thus, the input tree for the Cartesian index is a Cartesian tree that was created based on time-bound numerical data. In Figure 3.2, we can see an example of a Cartesian tree.

▶ **Definition 3.3** (Cartesian tree [32])**.** The *Cartesian tree* of an array is a binary rooted tree. The smallest element becomes the root, and the left and right subtrees are constructed recursively from the elements on the left or right side of the root in the array.

We found two papers [32, 33] in which the authors created algorithms to index Cartesian trees so that they do not lose information about the data patterns. Kim and Cho [32] proposed an algorithm that achieves a linear size index for the problem, which is, to the best of our knowledge, the most space-efficient solution.

There is another indexing problem that is similar in a way to Cartesian tree indexing. Žďárek and Melichar proposed a way to index 2D text that utilizes tree indexing [34, 35]. Similarly to the Cartesian index, the algorithm first creates a tree out of the 2D text, and the searching is done on that tree using pushdown automata.

3.1.3 Summary

During our research, we found the most common type of indexed trees to be labeled ordered trees. Not only many of the general tree solutions were intended for those trees, but also a lot of the specialized tree structures, such as XML and JSON, can be seen as specific types of labeled ordered trees. In the next section, we show the different types of query.

3.2 Query type

The second property of a tree indexing problem is the class of query patterns to which it requires the indexing structure to answer. The most straightforward type of query on tree-structured data would be a node. The simplicity originates from the fact that the indexing structure does not have to consider any of the surrounding nodes. However, when the surrounding nodes are irrelevant, it is unnecessary to interpret the data as trees, adding an unnecessary level of complexity; therefore, it is unsurprising that we have not found any node indexing structures. In the rest of the chapter, we focus on more complex query types.

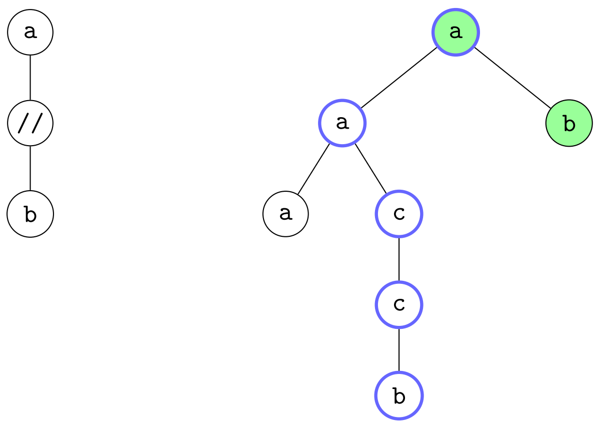


Figure 3.3 Let there be a query `/a//b` that can be represented by the diagram on the left. The "`/`" symbol in this context means that `a` has to be the root of the tree, and the symbol "`//`" signifies wildcard, meaning that any path can take its place. On the right is a tree over alphabet $\{a, b, c\}$ with highlighted paths onto which the query pattern matches. One match replaces the "`//`" node with path `acc`, the second match replaces the "`//`" node with nothing, directly connecting the root node labeled `a` with a node labeled `b`.

3.2.1 Path

The most rudimentary form of a query that is used in practice is a path. There are various ways to interpret a path query. Some indexes require the beginning of a path to match the root node of the subject data. Other techniques allow the query to begin at any node. Another type of path query might require the path to end in a leaf node. However, there is one unifying feature in all of the path indexes we found. It seems that all path query indexes match only onto paths that lay on a path from the root of an input tree to one of its leaves. For example, the indexes do not consider paths from one leaf to another or between sibling nodes.

Many XML documents take path queries as a fundamental building block for every other query by joining the result of path queries to allow branching. Furthermore, XML indexing often allows wildcards in path queries, which means that instead of a wildcard label, there can be a path with any number of nodes (even zero) on a path between the last defined node and the node right after the wildcard label. In Figure 3.3, we show an example of a path query with a wildcard and its potential matches.

There are many algorithms for indexing path queries [36, 37], most of them focusing on XML data using the XML Path Language [38, 24, 29]. However, we do not discuss path queries any further but rather show branching types of queries in the following sections.

3.2.2 Subtree

Subtree of a tree is a tree that has been made by picking a node and including all its descendants and related edges. All the leaves of a subtree match some of the leaves of the subject tree. A subtree query, in some sense, expands a path query by allowing branching.

With this definition of a subtree, there are at most n different subtrees in a tree of size n. Optimally, the search time does not depend on the number of subtrees or the size of an input tree. The most popular approach is to make the indexing time depend on the size of a query, searching time-wise, favoring shorter queries over longer ones. [4, 13, 2, 12, 14]

A very common problem within the subtree query category is the subtree indexing problem of labeled ordered trees [4, 13, 2, 12, 14]. The definition is as follows.

▶ **Problem 3.4** (Subtree indexing problem of labeled ordered trees)**.** Let there be a labeled

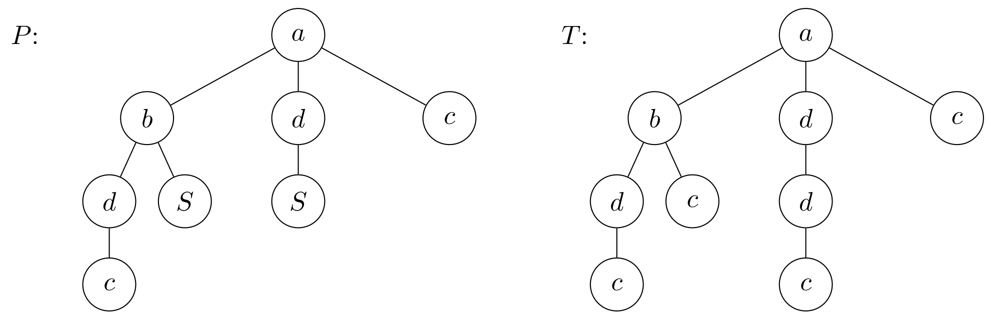


Figure 3.4 A tree pattern P over alphabet $\{a, b, c, d\}$, where rank(a) = 3, rank(b) = 2, rank(c) = 0, and rank(d) = 1, with an additional symbol S that is a wildcard for any subtree and an example of a tree T that P can matches onto.

ordered tree T. The *subtree indexing problem of labeled ordered trees* is to build an indexing structure that creates a complete index of T for its subtrees. This means that the index can answer whether a query in the shape of a tree is a subtree of T.

The subtree query precisely defines how the matches in an input tree look. When we need more flexibility, we can use tree pattern queries, which are discussed next.

3.2.3 Tree pattern

When we know the exact shape of what we are looking for, subtree indexing allows us to find the occurrences of the query in tree data. Unfortunately, that is not always the case. Sometimes, we may know only certain parts of how the occurrences are supposed to look. For those cases, there is tree pattern indexing. According to Trávníček et al. [11], such indexes benefit compiler code selection, interpretation of non-procedural languages, and implementation of rewriting systems.

A tree pattern query enables representing a tree with wildcards that match onto any subtree. The wildcards can only be in the leaves of a tree [13, 17, 39, 11, 10, 27]. In Figure 3.4, we see an example of a tree pattern.

▶ **Definition 3.5** (Tree pattern [11])**.** *Tree pattern* is a labeled ranked tree $T = (V, E)$ where the labeling is over a ranked alphabet A. Additionally, we use a special wildcard symbol $S \notin A$ with arity(S) = 0 as a placeholder for any subtree. A tree pattern must contain at least one node labeled by a symbol from A.

Although there are at most n different subtrees in a tree with n nodes, there are $2^{n-1}+n$ possible tree patterns [13, 17]. This fact often forces tree pattern indexes to have more sophisticated (complicated) solutions.

▶ **Problem 3.6** (Tree pattern indexing on ordered labeled trees problem)**.** Let T be an ordered tree. The *tree pattern indexing on ordered trees problem* is to build an index for T that gives the occurrences of subtrees that match a given tree pattern query.

The tree pattern indexing on ordered labeled trees problem definition describes the most common variant of tree indexing for tree pattern queries we came upon during our survey. There are many solutions to the problem, namely solutions for XML documents [27, 18] and automata solutions [13, 17, 39, 11, 10].

Although most of the tree pattern indexing solutions we found fit within the realm of arbology, there is no equivalent solution to the problem in stringology. When transformed into its linear notation, tree pattern matching can be seen as string matching with gaps [40]. However, the existing solutions of matching with gaps in stringology could not be translated directly into a tree indexing problem, and additional steps needed to be taken. [17]

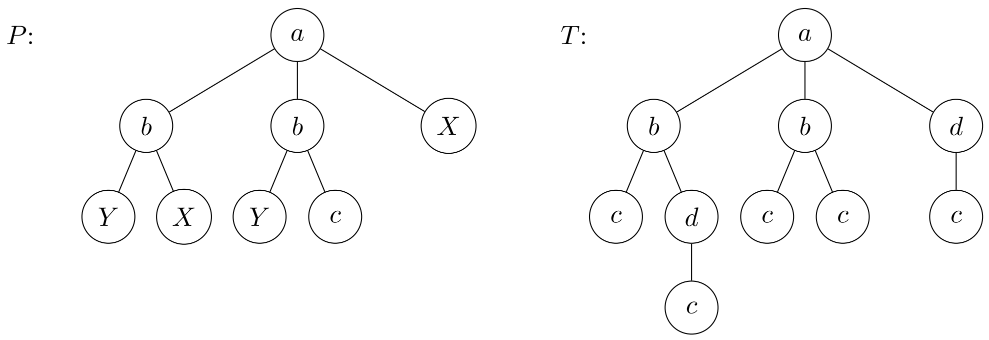


Figure 3.5 A nonlinear tree pattern P over alphabet $\{a, b, c, d\}$, where $\text{rank}(a) = 3$, $\text{rank}(b) = 2$, $\text{rank}(c) = 0$, and $\text{rank}(d) = 1$, with additional symbols Y, X that are wildcards for a subtree and an example of a tree T onto which P can match.

The solutions typically cannot offer the indexing time to depend solely on the query size, as is the case with subtrees. Some of the fastest solutions we encountered still had to consider the number of tree pattern occurrences in a subject tree. Formally, $O(m + \sum_{i=1}^{L} |\,\text{occ}(P_i)|)$, where m is the size of the pattern and $\text{occ}(P_i)$ are all occurrences of pattern P_i [13, 17, 39]. The algorithm is often composed of an automaton combined with a subtree jump table, which typically results in space complexity $O(n)$, where n is the size of the indexed data [17, 39].

3.2.3.1 Nonlinear tree pattern

For those cases, when a simple tree pattern index is too vague in defining the shape of a query, and we want to specify more about it, we have nonlinear tree pattern indexing. *Nonlinear tree patterns* are an extension of simple tree patterns as defined below. In Figure 3.5, we see an example of a nonlinear tree pattern.

▶ **Definition 3.7** (Nonlinear tree pattern [11]). *Nonlinear tree patterns* for a labeled ranked tree $T = (V, E)$, where the labeling is over a ranked alphabet A, do not have only one special symbol S, but instead can have many nonlinear variables $X, Y \ldots \notin A$. Similarly to S, all these symbols have an arity equal to zero. Contrary to S, every occurrence of the same symbol in a nonlinear tree pattern has to match onto the same subtree.

There are $(2 + v)^{n-1} + 2$ possible nonlinear tree patterns in a tree of size n, where v is the maximum number of nonlinear variables in a pattern. We have not found many solutions to this problem that focus on general trees (not only XML twig queries). The ones we have found need $O(n^2)$ space. [39, 11, 10]

3.3 Exactness of matching

The third property of an index defines what is considered a match and how lenient the index can be in its answers. Unless stated otherwise, tree indexing algorithms are implicitly *exact*. This means that indexes successfully accept and match only when a query pattern precisely matches parts of a given tree. Furthermore, exact indexes always give an accurate answer about the occurrences of the query matches.

However, exact indexes are not always attainable or even desired. Therefore, there exists a category of approximate indexing problems. During our research, we found two main reasons why approximate indexes are created. One of the reasons is that searching for exact matches with precise answers can be demanding on indexing time and memory consumption. Therefore, a trade-off of precision for higher effectivity has been made. The trade-off can result in an index

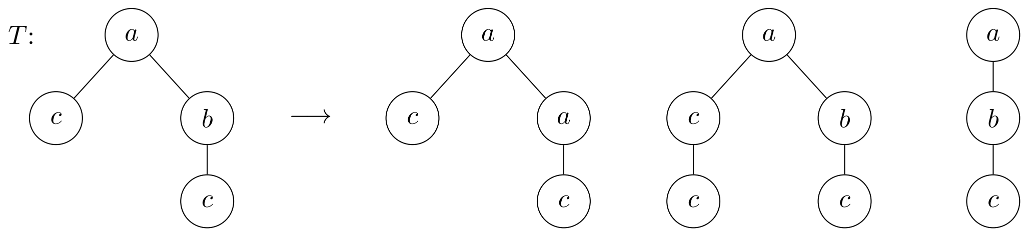


Figure 3.6 Diagram of a tree T and three trees created from T by applying certain operations. The tree on the left is created by renaming the node labeled b to a. The tree in the middle is created by inserting a node in T. The tree on the right is created by deleting a node from T.

that finds the exact match of a query but can give only approximate locations. Alternatively, it can cause the index to find a similar match to the query pattern, perhaps at the expense of not finding an exact match. Either way, the inaccuracy is seen as unfavorable. Examples of such indexes are subtree oracle PDA [12], and often the class of sequence-based indexes for XML data [19].

The second reason to create approximate indexes is to find the most similar matches to a query pattern. That is the case when we need to find some matches, even though they might not be exact. The index then has to prioritize the matches that are more similar to the ones that are less similar.

In the rest of this section, we show an example of an index that searches for patterns similar to a query pattern in a given tree. At first glance, this category might look analogous to the category where a query is in the shape of a tree pattern in that both query patterns only give a guideline to what a match in an input tree looks like. However, approximate subtree indexing is a very different problem, unrelated to indexing for tree patterns in both the definition of the problem and its solutions. When a query is in the shape of a tree pattern, we can precisely define which nodes will be in the answer subtrees as well as areas of a subtree match of which we do not know the shape. When indexing for approximate subtrees, we can indicate the general shape of the subtree and maybe even the number of alterations we allow. However, we do not know which query nodes will be altered, giving us less control over the outcome.

Typically, the operations that are allowed on a query are the following [4, 15]:

- the deletion of a node,
- the insertion of a node, and
- the renaming of the label of a node.

In Figure 3.6, we see various operations that can be done on a tree.

However, these alterations can be expanded, changed, or reduced to a different set. To further tailor the algorithm for specific data and their intended usage, we can value the operations differently, favoring one alteration over another.

Let us have two trees. Then their *edit distance* is the lowest possible cost of a sequence of operations necessary to alter the first tree into the second. To assign each operation on a tree a specific preference, we can use a *cost function*. It allows us to, for example, choose the insertion operation of a node over deletion when possible. A special kind of cost function is the *unit cost function*, which sets all the operation preferences equal. We have found a solution that uses the unit cost function [15]. We have yet to find a tree indexing technique that works with different cost functions.

▶ **Problem 3.8** (Approximate subtree indexing problem)**.** Let T be a tree. The *approximate subtree indexing problem* is to build an index for T that for each subtree query outputs the occurrences of the most similar matches. The similarity of the matches is calculated based on the operations needed to create the query pattern from the matched subtree.

Unfortunately, we have not found many indexing techniques that focus on the approximate subtree indexing problem. Therefore, we outline the features of only one tree index proposed by Cohen [15]. The index is meant for general trees, not bound to only one kind of data structure (such as XML). The solution offers additional specifications apart from a query pattern. The algorithm defines two variables, a similarity limit m and a quantity bound k. The variable m sets the maximum number of changes in the query. The variable k lets us choose the k most similar answers to the query.

In this section, we have shown the different options for what can be considered a match. In the next section, we discuss the different ways to report a found match.

3.4 Answer type

The last property of an index that we found crucial to the definition of a tree indexing problem is the type of expected answers. Many tree indexing solutions primarily offer one type of answer, although the algorithms can often be modified to fit some other type without significantly changing the solution. However, there are cases where the type of answer can substantially change the requirements of a problem's solutions.

In this chapter, we review the different types of answers. First, we overview the most common type of answer, the locations of all query matches. Next, we show what the word *first* can mean in the context of selecting only certain occurrences of a query pattern. We then describe an alternative type of answer that is binary, followed by an example of an indexing problem that utilizes it.

Perhaps because of the great variability of use when knowing the exact locations, the most common types of answers are the locations of the query matches within an input tree. Moreover, location answers are seen as the default in many indexing problems, not only in those specific to trees.

There are many options as to what we can consider the location of a query match. It can be, for example, the position of the root of a query pattern or its leftmost (rightmost) leaf. The important part is that we can unambiguously determine the occurrences of the query pattern in the tree data through the location. Answers indicating the locations of query matches are typical for XML document indexes and file systems.

Moreover, indexes that give answers in the form of locations can often be easily converted to give different types of answers. For example, to give the number of query matches. Unless stated otherwise, we expect a complete list of all the locations of a query pattern in the input data.

Some tree indexing problems might require only the first occurrences of a query pattern in a subject tree. The meaning of the word "first" changes in the different tree indexing problems. In exact tree indexing problems, the term *first occurrence* refers to a position within an input tree. For example, the answer is the first occurrence of a query in the preorder traversal of a given tree. In the approximate subtree indexing problem, the term *first occurrences* refers to the subtrees that are the most similar to a query pattern. For example, the index proposed by Cohen [15] allows a user to define the quantity bound k, meaning that the index structure gives the user the k most similar matches.

Another type of answer is a binary value that represents whether a query pattern matches a part of the input tree. This type of answer can allow modifications of solutions that reduce memory consumption. For example, an index might merge multiple representations of duplicate subtrees into one single representation, losing the distinct locations of the subtrees. Some *subtree rejection problem* solutions use such optimizations.

▶ **Problem 3.9** (Subtree rejection problem)**.** Let T be a labeled ordered tree. The *subtree rejection problem* is to build an index for T on which we can rely that when it answers that a query pattern is not present in T, it is always true.

In this section, we have shown the last category that helps define a tree indexing problem. From now on, we show the different approaches and properties of solutions to tree indexing problems.

3.5 Approaches to indexing

Now that we have discussed the different properties of a tree index, we outline the main approaches to the solutions. We do not consider the way a problem is solved to be a property of the indexing problem. Therefore, this section does not provide further information about the definition of a tree indexing problem but rather accompanying information about the possible algorithms.

As we could see earlier in the thesis, the word “tree” in tree indexing refers to the structure of the data in which we search. However, the indexing structure itself can take many different shapes, for example, tree, automaton, or table. In this chapter, we chose to describe two prevalent approaches to tree indexing problem solutions in more detail.

Although there are parallel algorithms for indexing texts [41, 42, 43, 44, 45, 46], we have not found many solutions to parallel tree indexing. Therefore, all the presented solutions to tree indexing problems are sequential.

3.5.1 Tree index structure

One of the more popular takes on tree indexing includes an indexing structure that is accompanied by an algorithm to create a full index. The indexing structure is frequently a tree that represents or extracts some of the properties of the subject tree data. The structures are often B-trees [1], namely B+-tree [18], R-trees, or their modifications [22, 30, 47, 26]. Examples of tree index structures include XR-tree [22, 30], B-tree [1], B+-tree [18], CB-tree [47], BF-tree [48], and C-tree [49, 29].

The typical process is to first create a tree indexing structure by preprocessing data. The nodes of the new tree structure do not generally have the previous labels as their sole name. Generally, the time it takes to create the indexing structure is long compared to the time needed to answer a query. It usually happens only once at the beginning. However, some index structures provide an interface for a quick update of the data, allowing for further changes. Once we have a complete index structure, querying can begin. A query might be transformed into a different representation depending on the algorithm. The algorithm, in some manner, traverses the index. Then it outputs an answer, all the query pattern occurrences in the initial data.

In this section, we have briefly discussed indexing structures in the shape of a tree. Next, we look at indexing structures in the form of pushdown automata.

3.5.2 Arbology

The term *arbology*, which refers to a way of approaching tree processing, was first proposed by Janoušek [4]. The general outline of arbology solutions to tree processing problems is as follows. First, we transform trees into their linear notation. Afterward, we use a PDA to solve a given problem. Having the linear representation of a tree means that we can essentially treat the structure as a string with some particular properties.

Tree processing problems have a finite pool of answers. For example, in subtree indexing, the index has to accept only a finite number of queries because there is only a finite number of subtrees in a tree. Even tree pattern indexes accept only a finite number of queries. Although a single tree pattern can match an infinite number of trees, there is only a finite number of tree patterns that match one whole tree. However, tree pattern indexes must also be able to match subtrees, not just a whole tree. Since there is always a finite number of subtrees in a tree, it still

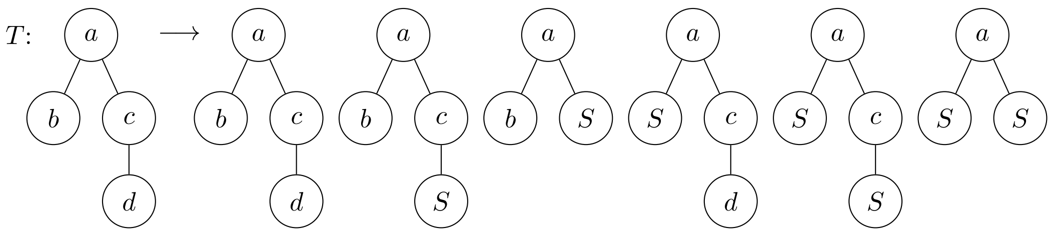


Figure 3.7 A tree T over alphabet $\{a, b, c, d\}$ and all tree patterns that T matches onto.

holds that tree pattern indexes match only a finite number of queries. In Figure 3.7, we can see an example of a tree and all the tree patterns that match it.

Should we translate the problem of accepting a finite number of queries into automata terminology, the accepted language is finite. It is common knowledge that every finite language can be represented by a finite automaton, leading us to believe that they are the optimal solution to those problems. However, according to arbology [4], there are three reasons that make pushdown automata the appropriate model of computation. Those reasons are listed below.

- The whole language group of linear notations of trees lies within the group of context-free languages, which means that there exists an equivalent PDA for it.
- Stringology solved many problems on strings. Their solutions mostly use finite automata. We can expand those stringology solutions to work with particular kinds of strings–trees in linear notation. In order for the process of expansion to be straightforward, we need the extra computational power of the pushdown store. If we applied the stringology algorithms to linear notations of trees without the pushdown modification, we would get the correct answers for the string that is a linear notation of a tree, but we would not get the correct answers for the underlying tree. For example, the string factor automaton for a prefix notation of a tree would accept all the substrings of the prefix notation, not only the substrings representing subtrees.
- Lastly, often traversing and manipulating trees requires recursion, which is easier to implement with pushdown automata rather than finite automata. For example, to compose the prefix notation of a tree, we can first write the label of the tree's root and then recursively treat the root's children as the new roots.

Additionally, should a tree index be implemented as a finite automaton, the index can have many more states [10].

One considerable drawback of pushdown automata is that not every nondeterministic PDA can be altered into a deterministic one. It is known that there are three main types of PDA that can be turned into equivalent deterministic PDA. Luckily, arbology solutions often produce input-driven pushdown automata, one of the classes of pushdown automata that can be made deterministic. [4]

Arbology offers a simple and unambiguous linear representation of trees. The representation does not need much more space than the number of nodes and is easily converted back into the tree itself.

▶ **Definition 3.10** (Prefix notation of a tree [4]). Let $T = (V, E)$ be a tree. The *prefix notation* of a tree, pref(T), is described by the two following rules:

- pref(T) $= v$ if $v \in V$ is a root without children,
- pref(T) $= v$ pref(w_1) pref(w_2)... pref(w_n), where v is the root of T and nodes $w_1, w_2 \dots w_n$ are its children.

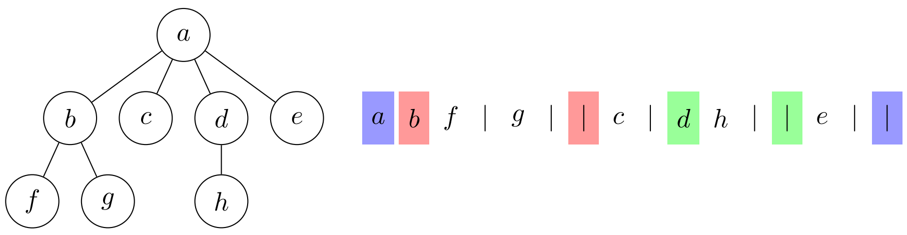


Figure 3.8 A labeled tree and its prefix bar notation.

▶ **Definition 3.11** (Prefix bar notation of a tree [4])**.** Let $T = (V, E)$ be a tree. The *prefix bar notation* of a tree, prefBar(T), is described by the two following rules:

- prefBar$(T) = v|$ if $v \in V$ is a root without children,
- prefBar$(T) = v$ prefBar(w_1) prefBar(w_2)... prefBar$(w_n)\,|$, where v is the root of T and nodes $w_1, w_2 \ldots w_n$ are its children.

The prefix bar notation is used when the arity of nodes is unknown. We can think of it as if each node has a bar linked to it, and the bar signifies that the subtree below its node has been completely described. Figure 3.8 has an example of a tree and its prefix bar notation.

Some notable indexing solutions using arbology techniques include subtree PDA [4, 13, 2, 14], subtree oracle PDA [12], tree pattern PDA [4, 17, 13, 39, 10], and nonlinear tree pattern PDA [39, 11, 10]. Some of the indexes are proposed in their nondeterministic versions, which are simpler to present and take up less space. Fortunately, the solutions tend to be an instance of input-driven PDA, which means that they can be made deterministic.

The tendency among subtree PDA and subtree oracle PDA is that the pushdown automata use only one pushdown store symbol. This allows the pushdown store to be implemented as a simple counter rather than as some container. [11]

Some of the techniques used in arbology solutions are not unique solely to arbology. This is especially true for rewriting trees into some form of an array (not necessarily into the prefix or prefix bar linear notation) with the combination of matching subsequences [28, 1, 16, 18].

3.6 Size of an index

Since indexing algorithms are intended to work with vast data, their size complexity is crucial. This section provides a general overview of what we can expect size-wise with the tree indexing problem variations. We focus on a few papers that take the size complexity as one of their main issues.

Quite a lot of indexing algorithms aim at a space complexity linear in the size of a subject tree, or at least polynomial [37, 17, 10, 13, 12]. The exponential size of an index (or larger) is typically not favorable as the tree data are usually extensive. Therefore, having an exponentially sized index built over the data is not realistic in many cases, as there is insufficient space.

However, space is a commodity in computer science and, therefore, we can exchange certain properties for larger space requirements [36, 47]. Since some of the acquired functionalities may not be available otherwise, it makes even exponential algorithms desirable [24, 13].

Furthermore, the space complexities we mentioned so far are always the worst case. Most of the linear space algorithms we came upon are truly linear in the size of an input tree in all cases. For example, the subtree oracle PDA has always got $n+1$ states, where n is the size of a subject tree [13]. However, the worst-case exponential algorithms do not have to be exponential in most instances. For example, Tree Paths Automata (TPA) space complexity is $O(h * 2^k)$, where h is the height of a tree and k is the number of leaves; however, experiments show that TPA is often linear in size to the subject tree [24].

There are solutions that ease off the typically large space requirements by offering compressive or compact indexing structures. Since space complexity is such an important factor, even algorithms that do not offer minimal possible space requirements make a note of being compressible [50].

One seemingly obvious approach is to translate an input tree into an array and then compress the array with one of many algorithms for string compression. Despite the vast number of string compression methods, only a handful of them can be used on an array while retaining the ability to index the tree data. [13]

As far as we know, the best results can achieve logarithmic size compared to the subject tree [13]. There are many papers dealing with this problem [13, 16, 32]. However, while we had only gone through a few of these publications, the solutions often seem to be specialized in a specific type of data.

3.7 Summary

In this chapter, we have shown different properties by which we can define tree indexing problems. The first property is the input tree type and it contains categories that include: ordered trees, labeled trees, and free trees. However, an index does not have to be created for general tree data; it can focus on one specific data structure, for example, XML documents. The second property is the query type, which often takes the shape of a path, a subtree, or a tree pattern. The third property defines whether the problems are searching for exact or approximate matches. However, the definition of what approximate means must be further specified by the tree indexing problem. The last property of an index is the type of answers it outputs. Most commonly, it is the locations of occurrences of a query in an input tree.

Toward the end of the chapter, we described two main approaches to the tree indexing problem solutions, arbology and its automata solutions, and tree structure indexes. Finally, we concluded the chapter with a summary of space complexities and indexes that focus on minimizing them.

Chapter 4
Subtree rejection problem

As mentioned in the previous chapter, the subtree rejection problem focuses on whether a query is not present in a subject tree rather than where. This means that the answer to a query in the subtree rejection problem is not the locations of the subtree matches or even the number of occurrences, but rather a simple answer *yes* or *no*. Additionally, we expect that only the answers that say that the query pattern is not present in the subject tree are reliable.

There are two main approaches to the problem. First, we can see it as an extra feature of the standard indexing techniques, as they inform us of all the locations of a query, but also, they should inform us when the query is not present. The second approach takes the rejection problem as its primary goal, often not offering the functionality of locating queries at all in exchange for better space complexity. In this chapter, we demonstrate solutions that take both approaches.

The presented solutions are not the only algorithms that solve the problem of subtree rejection. For general tree structures, the solution can be achieved using the tree compression automaton [13] or even the tree automaton. However, the querying time on tree automata depends not only on the size of a query but also on the size of a subject tree, which we generally try to avoid. Moreover, there are many more XML-specific solutions. For example, solutions that take the approach of seeing subtree rejection as a side effect of a full index are Xseq [27], PRIX [28], ViST [18], and Ctree [29]. Another XML-specific solution focusing on subtree rejection as its main target is presented in a psiX indexing system [21] designed for peer-to-peer databases. The authors of the solution conclude that rather than indexing for locations of a query right away, it is better to have a fast and space-efficient index that quickly recognizes whether the query is present in its documents.

Furthermore, many more categories of tree indexing can be applicable to subtree rejection. However, these are typically more advanced solutions with regard to both space complexity and search time complexity. Since the solutions are computationally unnecessarily demanding, it is better not to use them if the only use case we have in mind is subtree rejection. Nevertheless, should we combine the subtree rejection problem with another problem, those solutions are perfectly able to do both, for example, the whole category of tree indexing for tree pattern queries and approximate tree indexing combined with the subtree rejection problem.

The rest of the chapter is divided into four sections. First, since all three algorithms we present in this chapter use the prefix notation of trees, we explain the properties of subtrees of trees in their prefix notation. In the second section, we show subtree pushdown automata. In the third section, we offer an oracle modification of subtree pushdown automata called subtree oracle pushdown automata. In the last section, we present our own novel indexing technique based on the finite automaton called subtree finite automaton.

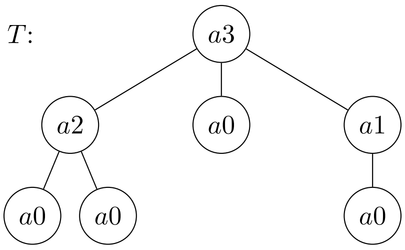


Figure 4.1 A ranked ordered tree T, where $\text{pref}(T) = a3\,a2\,a0\,a0\,a0\,a1\,a0$. Let us have two strings $x = a2\,a0\,a0$ and $y = a0\,a0\,a1$. The arity checksum of x is equal to $2 + 0 + 0 - 3 + 1 = 0$, which means that x is a subtree in its prefix notation. On the other hand, the arity checksum of y is equal to $0 + 0 + 1 - 3 + 1 = -1$ meaning that y is not a subtree in its prefix notation.

4.1 Properties of subtrees in prefix notation

All the solutions we discuss in this chapter use the prefix notation of a tree that was introduced in *Arbology* [4]. Therefore, we have decided to include this chapter, which states how trees and their subtrees relate to one another when written in the prefix notation.

When a tree is written in its prefix notation, we can treat it as a string. Searching for subtrees is, in that case, searching for factors in the string [14].

▶ **Theorem 4.1** (Substrings and subtrees in their prefix notation [14]). *Let us have a ranked ordered tree T. Then for each subtree S of T it holds that* $\text{pref}(S)$ *is a factor of* $\text{pref}(T)$.

The proof was given by Janoušek [4].

However, not every substring of $\text{pref}(T)$ represents a subtree of T. This can be demonstrated by the fact that there are at most n different subtrees in a tree of size n, but at most $(n^2 + n)/2$ substrings in a string of size n [14]. In order to distinguish substrings that represent subtrees from those that do not, we need the following definition and theorem.

▶ **Definition 4.2** (Arity checksum [14]). Let us have a ranked alphabet A and a string $x = x_1x_2x_3 \dots x_n$ over A, where $n \geq 1$. The *arity checksum* is denoted and defined by the formula $\text{ac}(x) = \text{arity}(x_1) + \text{arity}(x_2) + \text{arity}(x_3) + \cdots + \text{arity}(x_n) - n + 1 = \sum_{i=1}^{n} \text{arity}(x_i) - n + 1$.

▶ **Theorem 4.3** (Subtrees in their prefix notation [14]). *Let* $\text{pref}(T)$ *be a tree T in its prefix notation and x be a substring of* $\text{pref}(T)$. *Then, x is the prefix notation of a subtree of T, if and only if both of the following conditions hold,* $\text{ac}(x) = 0$ *and* $\text{ac}(x_1) \geq 1$ *for each x_1, where $x = x_1y$ and $y \neq \varepsilon$.*

The proof was given by Janoušek [4].

The above-presented theorem and definition together formulate rules for searching subtrees in the prefix notation of trees. In Figure 4.1, we use them to recognize a substring that represents a subtree from one that does not. The rules are utilized in all the solutions that we discuss next.

4.2 Subtree pushdown automata

Subtree pushdown automata represent complete indexes for ranked ordered trees, which means that they take the approach of treating subtree rejection as a secondary issue. These automata are analogous to string suffix automata from stringology. This is why we first state the main properties of the suffix automata.

The string suffix automaton is a finite automaton that represents a full index of a text for all possible suffixes. The answering phase is linear in the size of a query, which favors them over

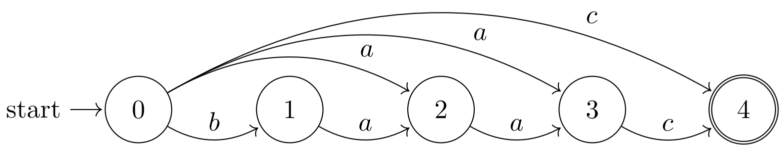


Figure 4.2 A suffix string automaton for string *baac*. The corresponding string suffix automaton is $(\{0,1,2,3,4\},\{a,b,c\},\delta,0,\{4\})$ with transitions illustrated in the diagram. The automaton is nondeterministic, however every nondeterministic FA can be turned into a deterministic one.

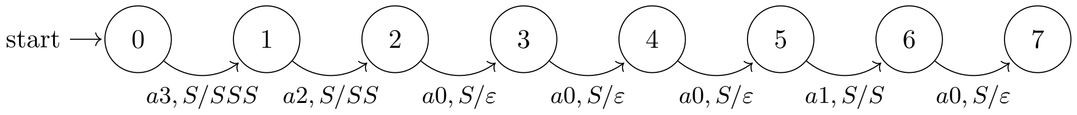


Figure 4.3 An automaton accepting the prefix notation of a ranked ordered tree. Let us have a tree whose prefix notation is $\text{pref}(T) = a3\,a2\,a0\,a0\,a0\,a1\,a0$. Algorithm 1 creates the following PDA $M_{\text{prefix}}(T) = (\{0,1,2,3,4,5,6,7\},\{a3,a2,a1,a0\},\{Z\},\delta,0,Z,\emptyset)$, where the transitions are as illustrated on the diagram.

pattern matching solutions, whose answering time are dependent on the size of the text, not the query pattern [14]. In Figure 4.2, we see an example of a suffix string automaton.

A tree of size n has at most n distinct subtrees. Similarly, there are n suffixes in a string of size n. The string suffix automaton does not work on trees in their prefix notation because once we state which node is the root of the subtree, we must include all of its descendants, but not all of its right siblings and their descendants, and possibly some other nodes. Therefore, the automaton must check when all of the node's descendants are included and then stop. The subtree PDA is designed to do that, which means that a subtree PDA can be created to make a full index for subtrees for any ranked ordered tree [14].

▶ **Definition 4.4** (Subtree PDA [14])**.** Let there be a tree T with its prefix notation $\text{pref}(T)$, then a *subtree PDA* for $\text{pref}(T)$ accepts all subtrees of T in their prefix notation by an empty pushdown store.

The construction of a subtree PDA is done in two phases. The first phase creates a nondeterministic subtree PDA, where the pushdown store computes the arity checksum that allows us to recognize subtrees [14]. The second phase does not add any functionality—it solely makes the automaton a deterministic one.

```
Algorithm 1: Construct a PDA accepting the prefix notation of a given tree.
  input  : a ranked ordered tree $T$ over alphabet $A$ in its prefix notation
           $\text{pref}(T) = a_1a_2a_3\dots a_n$, where $n \geq 1$
  output: a deterministic PDA $M_{\text{prefix}}(T) = (\{0,1,2,\dots,n\}, A, \{Z\}, \delta, 0, Z, \emptyset)$ that
           accepts $\text{pref}(T)$
1 for each state $i$, where $1 \leq i \leq n$ do
2     Create a new transition $\delta(i-1, a_i, Z) = \{(i, Z^{\text{arity}(a_i)})\}$, where $Z^0 = \varepsilon$.
```

Algorithm 1 comes from Janoušek [14]. In Figure 4.3, we can see an example of a pushdown automaton, created by Algorithm 1, which accepts the prefix notation of a tree.

The following algorithm creates a subtree PDA for a tree in its prefix notation. The algorithm is a modification of the previous one by adding transitions, which simulate that a subtree can start in all the tree's nodes and comes from Janoušek [14].

```
Algorithm 2: Construct a nondeterministic subtree PDA for a tree in its prefix notation.
    input : a ranked ordered tree $T$ over alphabet $A$ in its prefix notation
            $\mathrm{pref}(T) = a_1a_2a_3 \dots a_n$, where $n \geq 1$
    output: a nondeterministic subtree PDA $M_{\mathrm{ns}}(T) = (\{0,1,2,\dots,n\}, A, \{Z\}, \delta, 0, Z, \emptyset)$
1 First, we create a PDA that accepts $\mathrm{pref}(T)$ as in Algorithm 1:
2 for each state $i$, where $1 \leq i \leq n$ do
3     Create a new transition $\delta(i-1, a_i, Z) = \{(i, Z^{\mathrm{arity}(a_i)})\}$, where $Z^0 = \varepsilon$.
4 Second, we enable matching subtrees:
5 for each state $i$, where $2 \leq i \leq n$ do
6     Create a new transition $\delta(0, a_i, Z) = \delta(0, a_i, Z) \cup \{(i, Z^{\mathrm{arity}(a_i)})\}$, where $Z^0 = \varepsilon$.
```

▶ **Theorem 4.5** (Correctness of Algorithm 2 [14])**.** *The PDA created by Algorithm 2 for a tree T in its prefix notation is a subtree PDA $M_{\mathrm{ns}}(T)$.*

The proof was given by Janoušek [4]. In Figure 4.4, we give an example of a nondeterministic subtree PDA created by Algorithm 2.

Although not every nondeterministic PDA can be transformed into an equivalent deterministic automaton, the subtree PDA belongs to a group of acyclic input-driven automata. Therefore, it can be made deterministic using the following algorithm, which comes from Janoušek [14].

```
Algorithm 3: Transformation of an acyclic input-driven nondeterministic PDA into
an equivalent deterministic PDA
    input : an acyclic input-driven nondeterministic PDA
            $M_{\mathrm{ns}}(T) = (\{0,1,2,\dots,n\}, A, \{Z\}, \delta, 0, Z, \emptyset)$, where the ordering of its states
            is such that if $\delta(p, a, \alpha) = (q, \beta)$, then $p < q$
    output: a deterministic PDA $M_{\mathrm{ds}}(T) = (Q', A, \{Z\}, \delta', [0], Z, \emptyset)$ that is equivalent to
            $M_{\mathrm{ns}}(T)$
1  Let stack content, $\mathrm{sc}(q')$, where $q' \in Q'$, denote a set of strings over $\{Z\}$.
2  Initially, $Q' = \{[0]\}$, $\mathrm{sc}([0]) = \{Z\}$ and $[0]$ is an unmarked state.
3  while there are unmarked states in $Q'$ do
4      Select an unmarked state $q'$ from $Q'$ such that $q'$ contains the smallest possible
        state $q \in Q$.
5      for each input symbol $a \in A$ do
6          Add transition $\delta'(q', a, \alpha) = \{(p', \beta)\}$, where $p' = \{p : (p, \beta) \in \delta(q, a, \alpha)$ for all
            $q \in q'\}$.
7          if $p'$ is not in $Q'$ then
8              Add $p'$ to $Q'$ and create $\mathrm{sc}(p') = \emptyset$.
9          Add $\omega$ to $\mathrm{sc}(p')$, when $\delta'(q', a, \gamma) \vdash_{M_{\mathrm{ds}}(T)} (p', \varepsilon, \omega)$ and $\gamma \in \mathrm{sc}(q')$.
10     Set state $q'$ as marked.
```

In Figure 4.5, we show an example of a deterministic subtree PDA created by Algorithms 2 and 3.

When a state q' in the new PDA has $\mathrm{sc}(q')$ equal to $\{\varepsilon\}$ or $\{\varepsilon, \emptyset\}$, then we know that the outgoing transitions of that state can never be used and, therefore, can be eliminated. This might make some of the states inaccessible and hence safe to remove [12]. Inaccessible transitions and states emerge only after determinization. They can appear on a path to accepting a suffix of the prefix notation of a tree that does not represent a subtree in its prefix notation.

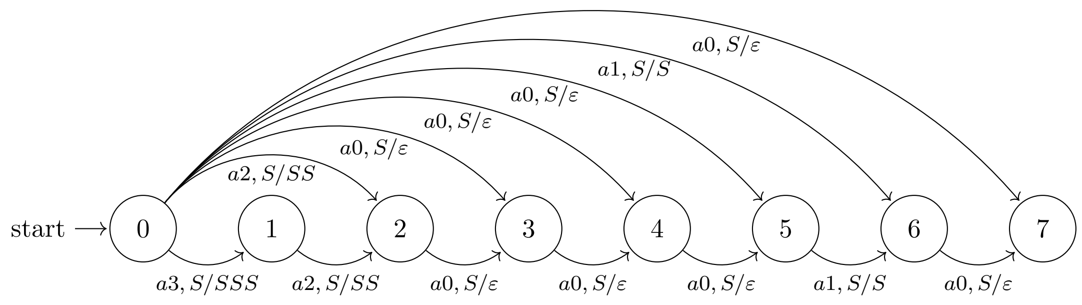


Figure 4.4 A nondeterministic subtree PDA for a ranked ordered tree T. Let the prefix notation of T be $\text{pref}(T) = a3\,a2\,a0\,a0\,a0\,a1\,a0$. Algorithm 2 creates the following subtree PDA $M_{\text{ns}}(T) = (\{0,1,2,3,4,5,6,7\}, \{a3, a2, a1, a0\}, \{Z\}, \delta, 0, Z, \emptyset)$, where the transitions are as can be seen on the diagram.

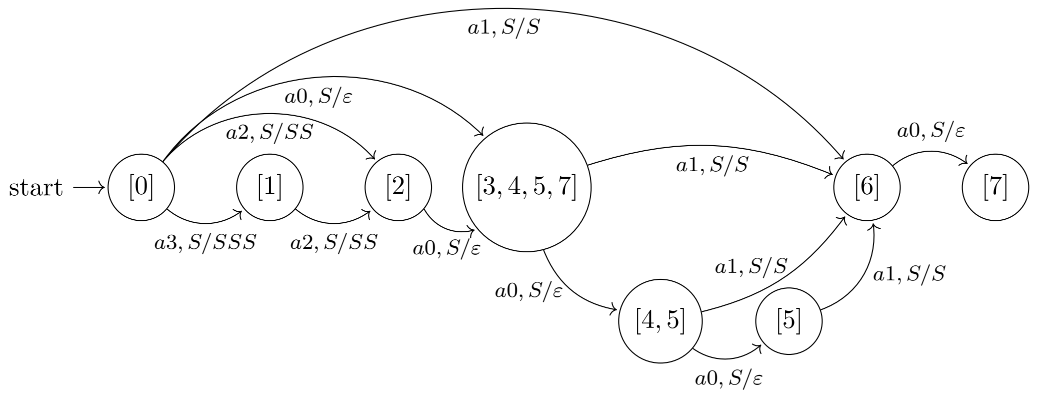


Figure 4.5 A deterministic subtree PDA for a ranked ordered tree T. Let the prefix notation of T be $\text{pref}(T) = a3\,a2\,a0\,a0\,a0\,a1\,a0$. Algorithm 3 transformed the PDA from Figure 4.4 into the following deterministic subtree PDA $M_{\text{ds}}(T) = (\{0,1,2,3,4,5,6,7\}, \{a3, a2, a1, a0\}, \{Z\}, \delta', [0], Z, \emptyset)$, where the transitions are as shown on the diagram.

▶ **Theorem 4.6** (Correctness of Algorithm 3 [14]). *Let us have an acyclic input-driven nondeterministic PDA $M_\mathrm{n}(T) = (Q, A, \{Z\}, \delta, q_0, Z, \emptyset)$, then a deterministic PDA $M_\mathrm{d}(T) = (Q', A, \{Z\}, \delta', [q]_0, Z, \emptyset)$ created by Algorithm 3 is equivalent to $M_\mathrm{n}(T)$.*

The proof was given by Janoušek [4].

Both nondeterministic and deterministic subtree pushdown automata use only one pushdown store symbol. Therefore, we do not have to implement a pushdown store but rather a simple counter.

▶ **Theorem 4.7** (Space complexity of subtree PDA [14]). *Let us have a tree of size n in its prefix notation. Then the deterministic subtree PDA constructed by Algorithms 2 and 3 uses only one pushdown symbol and has fewer than $2n + 1$ states and at most $3n$ transitions.*

The proof was given by Janoušek [14].

The query time complexity of subtree pushdown automata is $O(m)$, where m is the size of a query pattern, irrelevant to the size of an input tree [14]. This means that smaller queries are favored over larger ones, which is typically the case.

In this section, we discussed the properties and construction of subtree pushdown automata. We can create an oracle modification of subtree pushdown automata for better space efficiency, which we present next.

4.3 Subtree oracle pushdown automata

Subtree oracle pushdown automata take to the problem of subtree rejection a different approach from the above-mentioned subtree pushdown automata. Subtree oracle pushdown automata do not represent a full index for all subtrees, but rather promise to accept all the subtrees that are present in a given tree. Compared to subtree pushdown automata, they have less demanding space complexity while maintaining the same query time complexity $O(m)$, where m is the size of a query pattern [12].

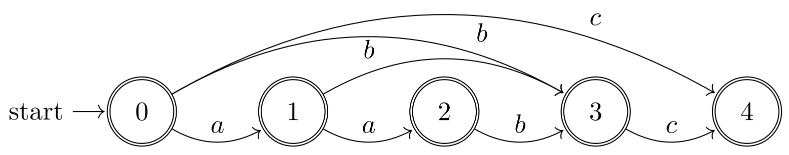
Subtree oracle pushdown automata are inspired by factor oracle automata from stringology. Therefore, in the following paragraphs, we give an overview of the properties of factor oracle automata.

Factor automaton is the minimal deterministic FA that accepts all factors in a given string. Factor oracle automaton is derived from the string factor automaton to optimize its space requirements. Factor oracle automaton made for a string of length n always has $n + 1$ states. It achieves reduced memory requirements by merging *corresponding states*. However, the factor oracle automaton can accept some subsequences of a string as well as all factors. [12]

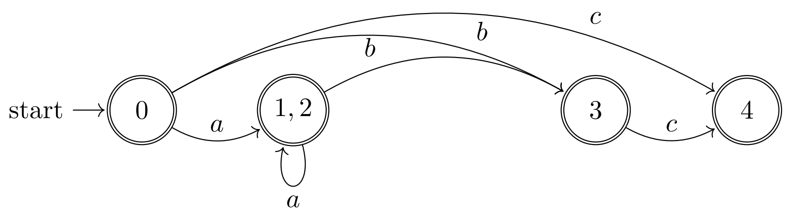
▶ **Definition 4.8** (Corresponding states [12]). Let there be a factor automaton for a string x and q, p, g are different states of the automaton. Let there exist two sequences of transitions: $(q, x_1) \vdash^* (p, \varepsilon)$, and $(q, x_2) \vdash^* (g, \varepsilon)$. If x_1 is a suffix of x_2 and x_2 is a prefix of x, then p and g are *corresponding states.*

However, we found Definition 4.8 of corresponding states to be imprecise. By merging the corresponding states as they were defined by Plicka, Janoušek, and Melichar [12], we can create an automaton that accepts more than only substrings and some subsequences. In fact, it can sometimes accept an infinite language. Moreover, the automaton can have less than $n+1$ states. This can happen when the string x_1 from the definition is handled earlier than x_2. Example 4.9 illustrates the problem.

▶ **Example 4.9** (The problem with Definition 4.8). Let us have a string $x = aabc$. Then the FA M_f in Figure 4.6 is the deterministic string factor automaton for x. By merging the corresponding states according to Definition 4.8, we create the factor oracle automaton M_fo in Figure 4.7. The two corresponding states are 1 and 2, as we can see in $(0, a) \vdash^* (1, \varepsilon)$, and



■ **Figure 4.6** A deterministic string factor automaton for *aabc*.



■ **Figure 4.7** The merging of corresponding states of the automaton from Figure 4.6.

$(0, aa) \vdash^* (2, \varepsilon)$. The problem with the resulting factor oracle automaton is that it has only n states and accepts an infinite language that includes strings that are neither factors of x nor its subsequences, for example, string aaa.

The same definition as Definition 4.8 can be found in *Text Searching Algorithms* [5]. This work offers the following interpretation of the definition.

> "Using our style of the numbering of states, the identification of corresponding states can be done as follows: Two states $p = [i_1 i_2 \dots i_{n1}]$, $q = [j_1 j_2 \dots j_{n2}]$ are corresponding states provided that set of states of nondeterministic factor automaton is ordered and the lowest states are equal which means that $i_1 = j_1$."

As we can see in Example 4.9, two corresponding states according to Definition 4.8 do not necessarily satisfy the requirement to begin with the same symbol. Therefore, the definition is not consistent with the interpretation of the definition. However, following the interpretation of the corresponding states, the factor oracle behaves as we expect it to, i.e., it always has $n+1$ states for a string of size n and accepts all factors of the string and possibly some of its subsequences. Therefore, in the rest of this text, we use the following definition for corresponding states in finite automata.

▶ **Definition 4.10** (Corresponding states [5])**.** Using ordered numbering of the states of the nondeterministic factor automaton, two states $p = [i_1 i_2 \dots i_{n1}]$, $q = [j_1 j_2 \dots j_{n2}]$ are corresponding when the lowest states are equal, which means that $i_1 = j_1$.

Alternatively, some authors, rather than defining corresponding states, show the algorithm to create a factor oracle automaton; see, for example, Crochemore et al. [51], Mancheron and Moan [52], Allauzen et al. [53], or Cleophas et al. [54]. Both definitions are equivalent in the automata resulting from them. We chose to go with the path of defining corresponding states to be consistent with the paper introducing subtree oracle pushdown automata [12].

▶ **Example 4.11** (Factor oracle automaton)**.** Let us have string *baac*. The corresponding factor oracle automaton is $(\{[0], [1], [2, 23], [3], [4]\}, \{a, b, c\}, \delta, [0], \{[0], [1], [2, 23], [3], [4]\})$ with transitions illustrated in Figure 4.9. It was created by merging corresponding states of the automaton shown in Figure 4.8.

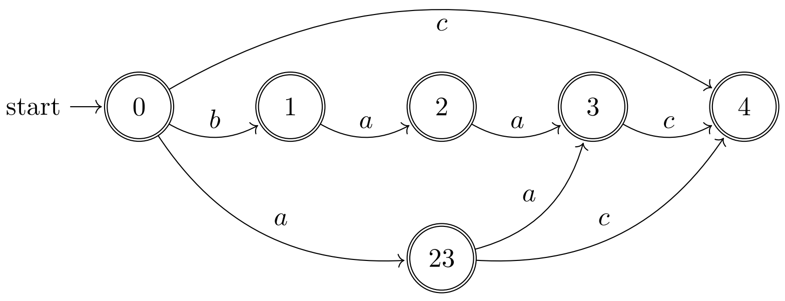


Figure 4.8 A deterministic string factor automaton for *baac*.

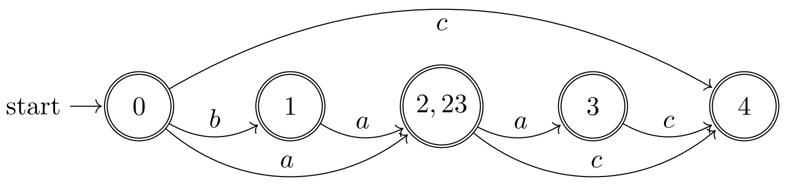


Figure 4.9 A factor oracle automaton for *baac* created by merging corresponding states of the automaton in Figure 4.8.

Subtree oracle pushdown automata work on both ranked and unranked ordered trees [12]. However, since subtree pushdown automata are originally presented on ranked trees, we demonstrate only the solution for ranked trees.

Subtree oracle PDA is an analogous solution to factor oracle automaton. Similarly, as the factor oracle automaton, the subtree oracle PDA has $n + 1$ states where n is the size of a tree. Furthermore, it is created by modifying the deterministic subtree PDA (presented in the previous section). The modification is to merge the corresponding states. [12]

▶ **Definition 4.12** (Subtree oracle automaton [12])**.** Let M_{ds} be a deterministic subtree PDA. *Subtree oracle PDA* is a PDA created from M_{ds} by merging all its corresponding states.

However, corresponding states were previously defined only for finite automata. Thus, the paper [12] that presents this solution offers an alternative definition.

▶ **Definition 4.13** (Corresponding states [12])**.** Let there be a subtree PDA for a tree T and q, p, g are different states of the automaton. Let there exist two sequences of transitions: $(q, x_1) \vdash^* (p, \varepsilon)$, and $(q, x_2) \vdash^* (g, \varepsilon)$. If x_1 is a suffix of x_2 and x_2 is a prefix of $\mathrm{pref}(T)$, then p and g are *corresponding states.*

For the same reasons as with Definition 4.8 of corresponding states defined on factor automata, we believe Definition 4.13 does not behave as intended. However, we cannot be quite sure of the meaning of Definition 4.13 because the transitions $(q, x_1) \vdash^* (p, \varepsilon)$, and $(q, x_2) \vdash^* (g, \varepsilon)$ omit the pushdown store content.

We presume that should we have an automaton $(Q, A, \{Z\}, \delta, q, Z, \emptyset)$ the meaning of the sequences of transitions is the following: $(q, x_1, Z) \vdash^* (p, \varepsilon, \gamma)$, and $(q, x_2, Z) \vdash^* (g, \varepsilon, \omega)$, where $\gamma, \omega \in \{Z\}^*$ and $p, g \in Q$. In other words, the stack contents are irrelevant.

Based on the previous presumption of the meaning of Definition 4.13, we conclude that after merging the corresponding states of a subtree PDA for the prefix notation of a tree T the resulting automaton can accept strings other than the expected subtrees of T in their prefix notation and some subsequences of $\mathrm{pref}(T)$. Furthermore, the automaton might be smaller than $n + 1$, where n is the length of $\mathrm{pref}(T)$.

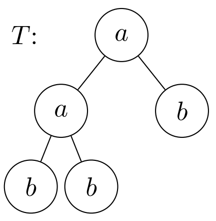


Figure 4.10 A ranked ordered tree.

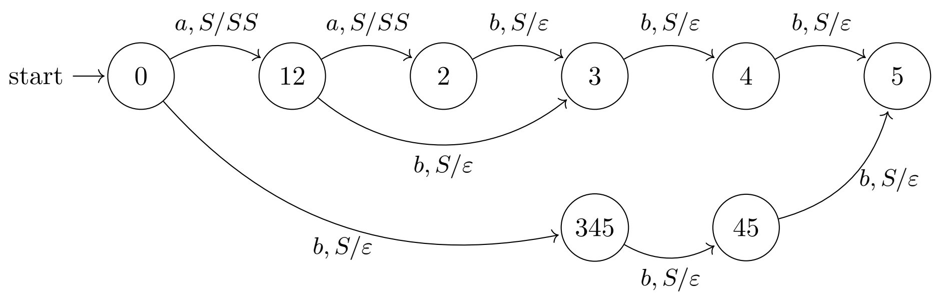


Figure 4.11 A deterministic subtree PDA for the tree in Figure 4.10.

▶ **Example 4.14** (Merging of corresponding states)**.** Let there be a tree T as illustrated in Figure 4.10, then the automaton M_{ds}, visualized in Figure 4.11, is the deterministic subtree PDA for T in its prefix notation. By merging the corresponding states as defined in Definition 4.13, we end up with an automaton M_{o1} as seen in Figure 4.12. As we can see, M_{o1} has fewer than $|\operatorname{pref}(T)| + 1$ states and accepts an infinite language. An automaton that has $|\operatorname{pref}(T)| + 1$ states and accepts only subtrees of T in their prefix notation and some subsequences of $\operatorname{pref}(T)$ is shown in Figure 4.13.

Therefore, we instead use Definition 4.10 of corresponding states. Although it is defined for finite automata, it is applicable for pushdown automata without change.

Furthermore, we must note that Algorithm 3 determinizing an acyclic input-driven PDA M_{ds} can create some inaccessible transitions and states. The inaccessible is recognized by stack contents sc, calculated by Algorithm 3. When a state q has its $\operatorname{sc}(q)$ equal to $\{\varepsilon\}$ or $\{\varepsilon, \emptyset\}$, then we know that all the outgoing transitions of that state are unusable. When the state has its $\operatorname{sc}(q) = \{\emptyset\}$, then we know that the state itself is inaccessible.

Let M_{io} be a PDA created by merging the corresponding states of M_{ds}, and M_{o} be a PDA created by first removing the inaccessible states and transitions from the automaton M_{ds} and afterward merging its corresponding states. The automata M_{o} and M_{io} are not necessarily equivalent. Some of the transitions and states of M_{ds} can be inaccessible because the pushdown

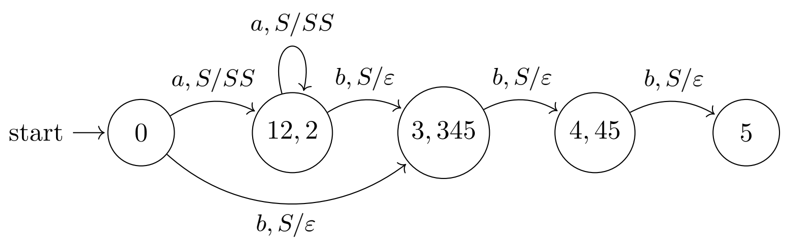


Figure 4.12 An automaton created by merging corresponding states of the automaton in Figure 4.11 according to Definition 4.13.

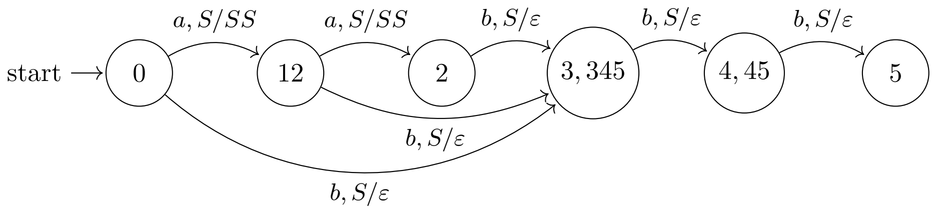


Figure 4.13 An automaton we expect to get as the result of merging corresponding states of the automaton in Figure 4.11.

store is empty before using the transition and entering another state. However, when we merge corresponding states, the inaccessible might get into a configuration that no longer always has an empty pushdown store, and therefore we can use the formerly inaccessible transitions and states. That leads to the fact that M_{io} can accept more strings than M_{o}.

▶ **Theorem 4.15** (Merging corresponding states of deterministic pushdown automata)**.** *Let M_{ds} be a subtree PDA that has been made deterministic by Algorithm 3, M_{io} be a PDA created by merging the corresponding states of M_{ds}, and M_{o} be a PDA created by first removing inaccessible transitions and states from M_{ds} and then merging its corresponding states. Then the relationship between the accepted languages of the automata is the following $L(M_{\mathrm{ds}}) \subseteq L(M_{\mathrm{o}}) \subseteq L(M_{\mathrm{io}})$.*

Proof. The proof falls naturally into two parts. We begin by proving that $L(M_{\mathrm{ds}}) \subseteq L(M_{\mathrm{o}})$ holds. For that, we first show that $L(M_{\mathrm{o}})$ always contains all of $L(M_{\mathrm{ds}})$.

Since M_{o} is created from M_{ds} in two steps, we must show that neither of the two stages deletes accepted strings. The first step is to remove inaccessible transitions and states. Inaccessible transitions and states never help to accept a word because we can never use them. A state is inaccessible when Algorithm 3 computes that the state has stack content equal to $\{\emptyset\}$. That means that there are no viable outgoing transitions and that we cannot accept any word in that state; therefore, we can never use such a state on a path to accepting configuration. A transition is inaccessible when the algorithm computes that the initial state of a transition has stack content equal to $\{\varepsilon\}$ or $\{\varepsilon, \emptyset\}$. Since every transition deletes one symbol from the top of the pushdown store before potentially adding some, it means that we cannot use this transition whatsoever. This shows that when we delete the inaccessible, we do not change the accepted language. When we merge two states, all that happens is that one new state has the ingoing and outgoing transitions of both of the two previous states. Some transitions of the two previous states might become the same when they are mapped onto a new state. However, that is the only way in which we can get a lower number of transitions than at the beginning. It is not because some transitions were deleted but rather because they got merged as well. Since that is the case for both transitions and states, we did not delete any words that the automaton accepts.

We showed that $L(M_{\mathrm{o}})$ contains at least $L(M_{\mathrm{ds}})$. Moreover, $L(M_{\mathrm{o}})$ can contain strings that are not in $L(M_{\mathrm{ds}})$. For example, let us have a ranked ordered tree T where $\mathrm{pref}(T) = a2\,a2\,a2\,a0\,a2\,a0\,a1\,a0\,a0\,a0$, then $M_{\mathrm{o}}(T)$ accepts string $a2\,a2\,a0\,a1\,a0\,a0$ while $M_{\mathrm{ds}}(T)$ does not.

Now, it remains to prove the second part, which we do by showing that $L(M_{\mathrm{o}}) \subseteq L(M_{\mathrm{io}})$ is true. We first show that $L(M_{\mathrm{io}})$ always contains all of $L(M_{\mathrm{o}})$. Let us assume that this is not true and that there is a word in $L(M_{\mathrm{o}})$ that is not in $L(M_{\mathrm{io}})$. That would mean that there is at least one transition (or even a state) in M_{o} that is not in M_{io} that is used on a path to accepting configuration. That would mean that the transition (and state) is not in M_{ds}, because M_{io} never deleted any states or transitions, only merged some. Therefore, M_{o} must have added the transitions (and states). However, that is a contradiction to how we created the automaton. Therefore, it cannot happen that a word in $L(M_{\mathrm{o}})$ is not in $L(M_{\mathrm{io}})$. Apart from strings from $L(M_{\mathrm{o}})$, $L(M_{\mathrm{io}})$ can contain other strings as is seen in Example 4.16. ◀

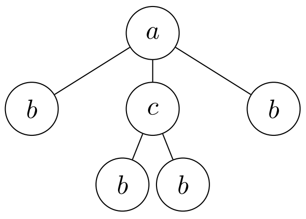


Figure 4.14 A ranked ordered tree.

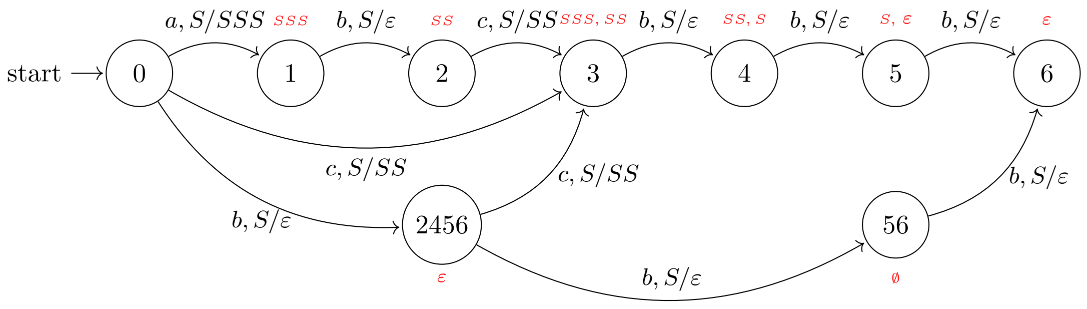


Figure 4.15 A deterministic subtree PDA for the tree depicted in Figure 4.14. Red labels show the stack content of states.

As a consequence of Theorem 4.15, to create a subtree oracle PDA that accepts the smallest language as well as all subtrees in their prefix notation, we must first remove all inaccessible states and transitions from a deterministic subtree PDA and only then merge its corresponding states.

▶ **Example 4.16** (Merging corresponding states of deterministic pushdown automata)**.** Let us have a tree T, which is illustrated in Figure 4.14, and its prefix notation is $\text{pref}(T) = abcbbb$. During determinisation of subtree PDA for that tree we create a PDA $M = (\{0, 1, 2, 3, 4, 5, 6, 2456, 56\}, \{a, b, c\}, \{Z\}, \delta, q_0, Z, \emptyset)$ in Figure 4.15 with two transitions—$(56, \varepsilon) \in \delta(2456, b, Z)$ and $(6, \varepsilon) \in \delta(56, b, Z)$, and one state—56 inaccessible. Figure 4.16 shows a PDA M_1 created by merging the corresponding states of M. Figure 4.17 illustrates a PDA M_2 created by first removing inaccessible transitions and states and then merging the corresponding states of M. Automata M_1 and M_2 accept a different language. For example, the string $abbb$ is accepted by M_1 while M_2 rejects it.

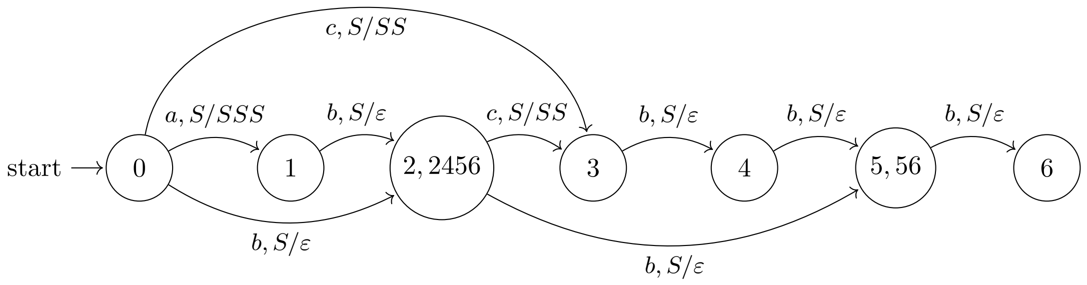


Figure 4.16 A subtree oracle PDA created by merging all the corresponding states of the automaton in Figure 4.15.

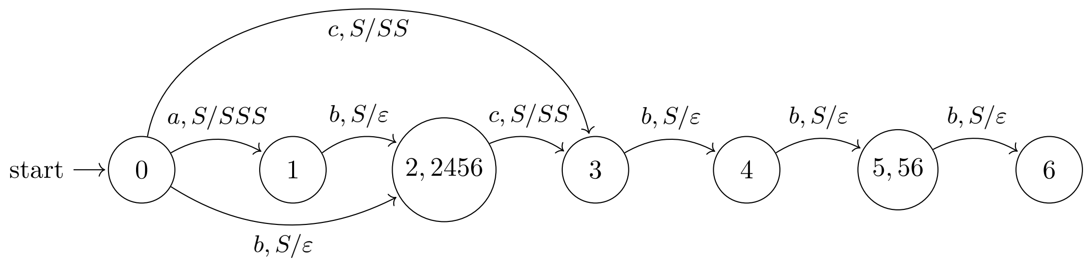


Figure 4.17 A subtree oracle PDA created from the automaton in Figure 4.15 by first deleting all the inaccessible states and transitions and only then merging the corresponding states.

Subtree oracle PDA, correspondingly to factor oracle automaton, accepts some strings that do not represent subtrees in their prefix notation. Those strings are a subsequence of the prefix notation of a tree [12]. Hence, while we can rely on the fact that rejected strings do not represent subtrees of a tree, we have to check every accepted string to see if it is truly present in the tree by some additional mechanism.

Furthermore, Plicka, Janoušek, and Melichar [12] define *safe merging*. The safe merging of states does not change the accepted language of an automaton. Properties of automata created by the safe merging of their states are unknown and therefore open to future research. [12]

Both subtree pushdown automata and subtree oracle pushdown automata discussed in this chapter use pushdown store operations. Should we avoid working with pushdown stores, we can use subtree finite automata presented in the next section.

4.4 Subtree finite automata

In this section, we propose a novel index for all subtrees based on a finite automaton called subtree finite automaton. Subtree finite automata represent the complete index of ranked ordered trees, which means that they also inform us when query patterns are not present in the trees; hence, they solve the subtree rejection problem.

The subtree finite automata are based on the fact that there is always only a finite number of subtrees in a tree, and therefore, the index has to accept and successfully answer only a finite number of queries. Finite languages belong within the group of regular languages, which are recognized by finite automata. Therefore, the additional computational power of a pushdown store is not vital to the index.

Although the space complexity of subtree finite automata can be higher by an order of magnitude than the space complexity of a subtree pushdown automata, we believe that subtree finite automata are a meaningful construct. The searching time of a subtree FA is $O(m)$, where m is the size of a query, and the size complexity is $O(n^2)$, where n is the size of an input tree.

Moreover, there is not only a finite number of subtrees in a tree but also of both linear and nonlinear tree patterns. However, the number of possible (non)linear tree patterns in a tree is exponential in the size of the input tree. We presume that this leads to a high space complexity of the potential FA.

To create a subtree FA for a ranked ordered tree T, we need to acquire all subtrees of T. The following definition represents that in the algorithm that creates a subtree FA.

▶ **Definition 4.17** (subtreeSet[1])**.** Let there be a tree T and pref(T) its prefix notation. Then subtreeSet(pref(T)) denotes the set of all subtrees of T in their prefix notation.

[1]The definition is based on a supervisor's idea

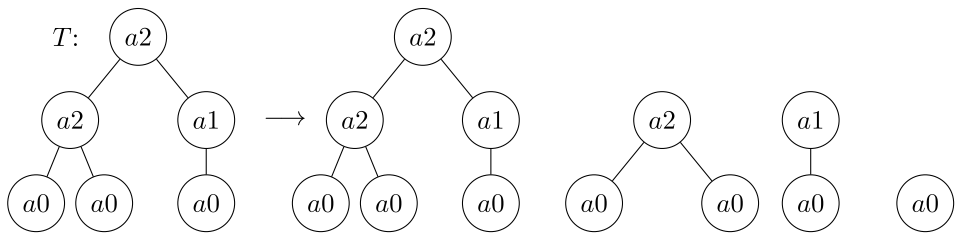


Figure 4.18 A ranked ordered tree T and all of its subtrees.

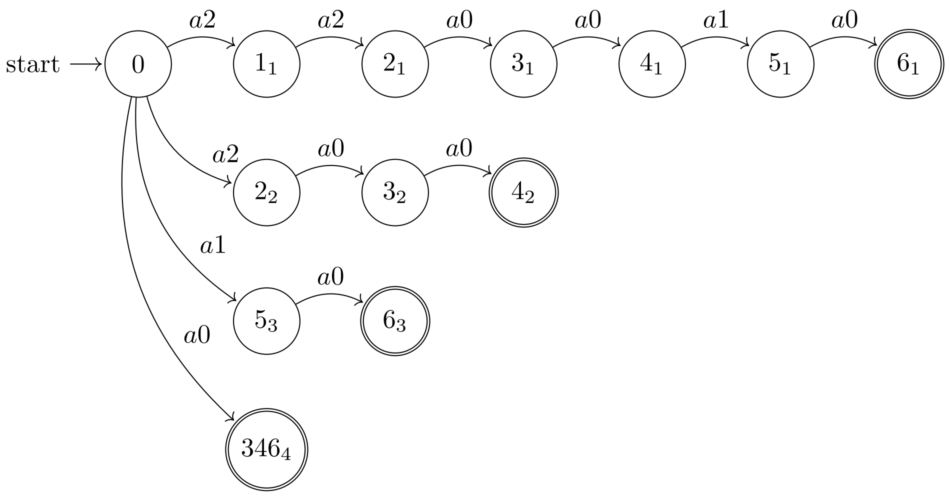


Figure 4.19 A subtree FA for the tree in Figure 4.18.

```
Algorithm 4: Construct a nondeterministic subtree FA for a tree in its prefix notation.
    input  : a ranked ordered tree $T$ over alphabet $A$ in its prefix notation pref($T$)
    output: a nondeterministic subtree FA $M_{\text{ns}}(T) = (Q, A, \delta, q_0, F)$
1 Let $q_0 = 0$.
2 for each string $x = a_1 a_2 a_3 \dots a_n$ in subtreeSet(pref($T$)) do
3     Let $j$ be a unique identifier of $x$.
4     Create $n$ states labeled $1_j \leq i_j \leq n_j$.
5     Create a transition $\delta(q_0, a_1) = \delta(q_0, a_1) \cup \{1_j\}$.
6     for each $i_j \in \{2_j, 3_j, \dots, n_j\}$ do
7         Create a transition $\delta((i-1)_j, a_i) = \{i_j\}$.
8     $F = F \cup \{n_j\}$.
```

Let us call a *branch* each move leading from the initial state of an automaton. Since on line 5 in Algorithm 4, we create one branch for each subtree of T, the nondeterministic subtree FA for T has its number of branches equal to $|\text{subtreeSet}(\text{pref}(T))|$.

▶ **Example 4.18** (Nondeterministic subtree finite automaton)**.** Let us have a tree T and its prefix notation $\text{pref}(T) = a2\,a2\,a0\,a0\,a1\,a0$. Then $\text{subtreeSet}(\text{pref}(T))$ is equal to $\{a2\,a2\,a0\,a0\,a1\,a0, a2\,a0\,a0, a1\,a0, a0\}$. In Figure 4.18, we can see T and all its subtrees from $\text{subtreeSet}(\text{pref}(T))$. Then the automaton $M_{\text{ns}} = (\{0, 1_w, 2_w, 3_w, 4_w, 5_w, 6_w, 2_x, 3_x, 4_x, 5_y, 6_y, 346_z\}, \{a2, a1, a0\}, \delta, 0, \{6_w, 4_x, 6_y, 346_z\})$, where the transitions δ can be seen in Figure 4.19, is the subtree FA created for T by Algorithm 4.

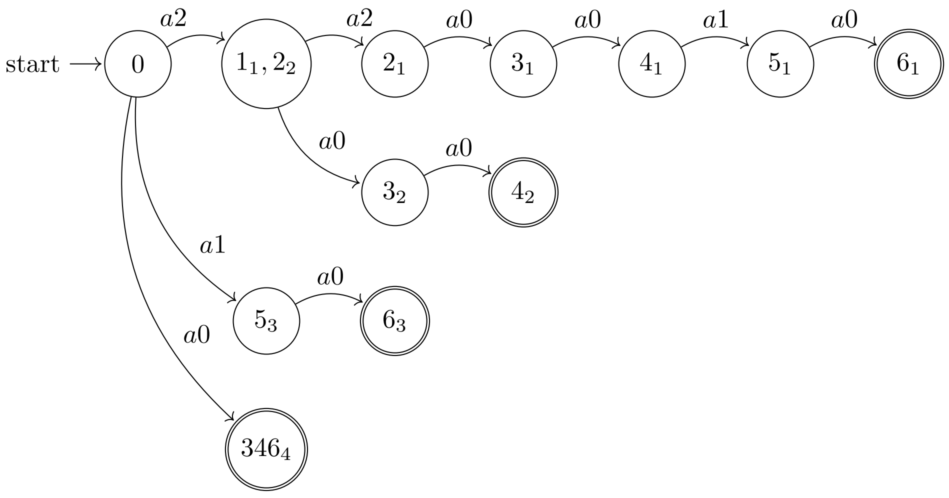


Figure 4.20 A deterministic subtree FA for the tree in Figure 4.18.

The nondeterministic subtree FA can be made into an equivalent deterministic automaton by the standard algorithm for the determinization of nondeterministic finite automata. The number of branches in a deterministic subtree FA (Q, A, δ, q_0, F) is less than or equal to $|A|$.

▶ **Example 4.19** (Deterministic subtree FA)**.** Let us have a tree T and its prefix notation $\text{pref}(T) = a2\, a2\, a0\, a0\, a1\, a0$. Then $\text{subtreeSet}(\text{pref}(T))$ is equal to $\{a2\, a2\, a0\, a0\, a1\, a0, a2\, a0\, a0, a1\, a0, a0\}$. Automaton $M_{\text{ds}} = (\{0, 1_w 2_x, 2_w, 3_w, 4_w, 5_w, 6_w, 3_x, 4_x, 5_y, 6_y, 346_z\}, \{a2, a1, a0\}, \delta, 0, \{6_w, 4_x, 6_y, 346_z\})$ was created by determinization of the automaton M_{ns} from Example 4.18. The transitions δ can be seen in Figure 4.20.

We can use Algorithm 5 when we need a deterministic subtree FA. Algorithm 5 creates a deterministic subtree FA directly, unlike Algorithm 4, whose resulting automaton must be transformed into a deterministic one.[2]

```
Algorithm 5: Construct a deterministic subtree FA for a tree in its prefix notation.
    input  : a ranked ordered tree $T$ over alphabet $A$ in its prefix notation $\text{pref}(T)$
    output: a deterministic subtree FA $M_{\text{ds}}(T) = (Q, A, \delta, q_0, F)$
1  Let $q, p$ be states of $M_{\text{ds}}(T)$, initially $q = q_0$.
2  for each string $x = a_1 a_2 a_3 \dots a_n$ in $\text{subtreeSet}(\text{pref}(T))$ do
3      for each $i \in \{1, 2, \dots, n\}$ do
4          if there is a transition $\delta(q, a_i) = p$ then
5              $q = p$
6          else
7              Create a state $p$ and $\delta(q, a_i) = p$.
8              $q = p$
9          if $i = n$ then
10             $F = F \cup \{p\}$
```

▶ **Theorem 4.20** (The space complexity of subtree finite automata)**.** *The space complexity of a subtree FA M for a given tree T with size n is $O(n^2)$.*

Proof. The more distinct subtrees T has, the more branches its subtree FA M has. T can have at most n distinct subtrees. Without loss of generality, let us assume that each of the prefix

[2]The author of the thesis designed the algorithm based on a supervisor's idea.

notations of the n distinct subtrees begins with a different label. Then, the most branches the subtree FA M can have is n. The largest subtree of a tree that is not equal to the tree is one, where we remove only one node, the root. Therefore, we obtain the largest subtrees possible by removing only one node in each iteration. A tree in the shape of a path, where one end of the path is the tree's root, meets the conditions of having the largest possible subtrees. Without loss of generality, let us assume that T is in the shape of a path. Then the sizes of its subtrees are $n, n-1, n-2, \dots, 2, 1$. There is one transition and one state for each node in a subtree. Therefore, each branch of the subtree FA M has the same number of states as the size of its corresponding subtree. That makes it $\Sigma_{i=1}^{n} i$ states and transitions. Furthermore, the subtree FA M has an additional initial state. Formally, $1 + \Sigma_{i=1}^{n} i$ states and $\Sigma_{i=1}^{n} i$ transitions. Therefore, T has at most $1 + \frac{n^2+n}{2}$ states and $\frac{n^2+n}{2}$ transitions. ◀

Subtree FA created by Algorithms 4 and 5 is not necessarily the minimal automaton that accepts the required language. It can be, like all finite automata, minimized. However, minimizing the FA could erase the ability to locate a subtree within the subject tree. Nevertheless, minimizing a subtree FA could be a valid step in some oracle modification of the automaton. However, that is open to further research.

Based on the conclusions drawn in the proof of Theorem 4.20, we expect subtree FA to work with the lowest relative space requirements on wide trees with lots of repetition. The least amount of necessary states, except for a tree in the shape of a single node, is $n + 1 + 1$, which is obtained, for example, by a tree of size n in the shape of a root whose children are all leaves with the same label. On the other hand, we presume that the automaton performs worse in space complexity on deep narrow trees for the same reasons as mentioned in the proof of Theorem 4.20.

Chapter 5

Implementation

In this chapter, we describe our C++ implementation of three solutions to the subtree rejection problem presented in Chapter 4; that is, subtree pushdown automata, subtree oracle pushdown automata, and subtree finite automata. We then show how we collected data for our experiments to assess and compare the efficiency of the three algorithms.

The rest of the chapter is organized as follows. First, we overview our implementation of the indexes. We describe how we implemented pushdown automata and the two solutions that use them: subtree pushdown automata and subtree oracle pushdown automata. Then we show our implementation of finite automata and the indexing algorithm subtree finite automata. Finally, we state how we collected data during the experiments.

5.1 Indexes

The central object of the whole program is the subtree rejection index. The object is represented by the `CSubtreeRejectionIndex` abstract class. Each of the implemented solutions to the subtree rejection index (indirectly) inherits from the class. There are two methods that all subtree rejection indexes have to implement. These methods are:

```
virtual void BuildIndex ( const std::string & tree ) = 0;
virtual bool Query      ( const std::string & queryPattern ) = 0;
```

The pure virtual `BuildIndex` method represents the preprocessing of a given tree to create its subtree rejection index. The aim of the `Query` method is to answer with a boolean value whether a concrete implementation of a subtree rejection index has found a match to a query pattern within an input tree.

For simplicity, the program accepts only trees labeled with a uniquely ranked alphabet. Both methods expect trees in their prefix linear notation (see Definition 3.10). The linear notation is represented by a string of whitespaces and integers. Each integer represents a node. The value of an integer is the arity of the corresponding node. For example, the methods accept trees such as the one illustrated in Figure 5.1, whose prefix linear notation is `2 0 1 0`.

All three solutions expect a valid input. We can afford this since the implementation of the algorithms is intended only for experiments, which saves some time not only in the preprocessing phase but also in each query call.

All the implemented solutions are automata, and the interpretation of a state of an automaton is the same across all the indexes; therefore, we created one class to represent a state of both finite and pushdown automata, called `CState`. The class represents the name of a state by a set of integers `std::set<int>`. Furthermore, the class contains a method called `AreCorresponding`

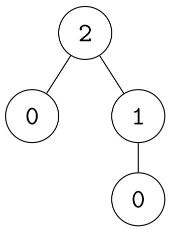


Figure 5.1 A ranked ordered tree with integers for labels.

that implements the logic behind determining whether two states are corresponding according to Definition 4.10.

Now we have overviewed the common parts of all the indexes. Next, we focus on our implementation of pushdown automata, which is the basis for subtree pushdown automata and subtree oracle pushdown automata indexes.

5.1.1 Pushdown automata

Both subtree pushdown automata and subtree oracle pushdown automata use only one pushdown store symbol. When that is the case, we can represent the pushdown store of a pushdown automaton by a simple integer counter instead of some container. In the implementation, we call these automata *counter automata* and represent them by the abstract class `CCounterAutomata`.

As for the structure of `CCounterAutomata`, it consists of only the data necessary to represent a PDA. In order to do so, `CCounterAutomata` defines two supporting structures `TFrom` and `TTo`, which are used to represent the transitions of an automaton. `TFrom` contains an instance of the `CState` class, an integer representing an input symbol, and an integer representing what ought to be extracted from the counter representing the pushdown store (the value is always set to 1 in the implemented solutions). `TTo` also contains an instance of `CState` and an integer representing what is added to the counter. We can think of the two structures as if `TFrom` instances are arguments of δ and sets of `TTo` instances are its results: $\text{TTo} \in \delta(\text{TFrom})$.

The 7-tuple that defines a PDA is represented in the following way. The states of an automaton are implemented as a set of states `std::set<CState>`. The alphabet is represented by a set of integers `std::set<int>`. There is no attribute to represent the pushdown store alphabet because there is only one symbol in it, and we do not need to know the exact symbol. Transitions are represented by a map from `TFrom` to a set of `TTo`, such that `std::map<TFrom, std::set<TTo>>`. The initial state is represented by an instance of the `CState` class. The initial pushdown store symbol is not represented because the specific pushdown store symbol is unnecessary since we know that there is only one symbol in the pushdown store alphabet; the important part is that there is always only one symbol initially in the pushdown store. Furthermore, we do not represent the set of final states because both of the implemented pushdown automata accept input strings by an empty pushdown store (in our case, the counter set to 0). To sum it up, from the 7-tuple that defines a PDA, we need to represent only four of the items because we implement counter automata accepting by an empty pushdown store.

Both `std::map` and `std::set` that we use in our implementation are usually implemented by red-black trees, which means that operations insert, find, and erase have $O(\log(n))$ time complexity, where n is the size of one of those two containers [55, 56].

Most importantly, the `CCounterAutomata` class provides two functionalities: to run a query and to make a PDA deterministic. Even though the `CCounterAutomata` class is not an index on its own, it inherits from `CSubtreeRejectionIndex`. Since running a query is the same across all counter automata, `CCounterAutomata` implements the `CSubtreeRejectionIndex::Query` method. The method reads one integer from the argument query pattern at a time. If the number of viable transitions, given the input symbol, is not equal to one, the method returns false. If a move to another state cannot happen due to the lack of symbols in the pushdown store (counter), the method returns false. Otherwise, when there are no other input symbols, the method checks whether the pushdown store is empty (the counter is set to zero) and returns the result. In conclusion, the Query method is expected to run on a deterministic pushdown automaton, and running it on a nondeterministic automaton can wrongly output a negative answer.

As for the time complexity of the Query method, we did not obtain the expected $O(m)$, where m is the size of a query pattern [12, 14]. In our implementation, we use the find method on the std::map to search for viable transitions. The find operation is logarithmic in the size of the container; therefore, in our case, it is logarithmic to the number of transitions of an automaton, resulting in overall time complexity $O(m \cdot \log(t))$, where m is the size of a query pattern and t is the number of transitions.

Furthermore, similar to querying, making input-driven acyclic pushdown automata deterministic is the same across all counter automata. Therefore, the CCounterAutomata class has a Determinize method that implements Algorithm 3. The method also deletes the automaton's inaccessible states and transitions, apart from making it deterministic.

Subtree pushdown automata indexes are represented by the CSubtreePDA class. The class inherits from CCounterAutomata and implements the CSubtreeRejectionIndex::BuildIndex method. The method implements Algorithm 2, which constructs a subtree PDA for a given tree and calls the Determinize method. For each tree, the method needs to be called once.

Subtree pushdown automata not only provide a binary answer on whether a subtree query is present in an input tree, but they can also tell where to find the occurrences. We can acquire the locations because of the following properties. To get into any one state of a subtree PDA, we always move through transitions that contain the same symbol on the input. The same holds for our implementation of a subtree PDA. All states of CSubtreePDA have a name in the form of a set of integers. These integers correspond to the locations in an input tree where we can find nodes with a label equal to the input symbol with which we got into a state. For example, let us have a state of an automaton labeled $\{4, 6, 12\}$ into which we can only get through transitions that have as an input symbol a. Then the corresponding nodes in the input tree that the state represents are at positions 4, 6, and 12, and their label is a. However, in the implementation, we do not utilize this property of the index since we build subtree rejection indexes that need only binary answers.

Subtree oracle pushdown automata are implemented as a COraclePDA class. The COraclePDA class inherits from CSubtreePDA and implements CSubtreeRejectionIndex::BuildIndex where it creates a subtree PDA and then merges its corresponding states; lastly, it calls the Determinize method. Within the BuildIndex method, we rename the states of the subtree PDA; therefore, we cannot obtain the location of a subtree in the input data tree.

5.1.2 Deterministic finite automata

Finite automata pose a similar role as counter automata in the implementation in that they are not an index themselves, but instead, they build the structure of an automaton. Deterministic finite automata are represented by a CDeterministicFA abstract class that inherits from CSubtreeRejectionIndex.

Similarly to CCounterAutomata, it defines two supporting classes, TFrom and TTo, to help represent transitions of an automaton $\delta(\text{TFrom}) = \text{TTo}$. Both TFrom and TTo contain an instance of CState, and TFrom additionally contains an integer representing a symbol on the input.

In order to represent a finite automaton, CDeterministicFA contains four member variables. Two variables that represent all the states of an automaton and all the final states, both are of type std::set<CState>. To represent the transitions of an automaton, it contains a member variable of type std::map<TFrom,TTo>. To represent the initial state, it contains an instance of CState. We do not need to represent the alphabet of a finite automaton since subtree finite automaton does not need it during the building phase (unlike pushdown automata indexes that need it for determinization).

`CDeterministicFA` provides the implementation of the `CSubtreeRejectionIndex:Query` method. The method returns `false` when there is no viable transition on the current state and input symbol or when the automaton ends in a non-final state. Regarding time complexity, the implementation of the `Query` method faces the same drawback as in `CCounterAutomata`. Because we use the `find` method on a `std::map` the time complexity is $O(m \cdot \log(t))$ instead of $O(m)$, where m is the size of a query pattern and t is the number of transitions.

Analogously to subtree pushdown automata, subtree finite automata indexes are represented by the `CSubtreeFA` class that inherits from `CDeterministicFA` and implements the `CSubtreeRejectionIndex::BuildIndex` method. In the `BuildIndex` method, we implement Algorithm 5. To acquire the subtreeSet(pref(T)) of a tree T, we implemented a `subtreeSet` method that for each node of T finds a subtree that treats the node as its root. In order to find the prefix notation of each subtree of T, we calculate the arity checksum beginning in the temporary root node, continuing until the arity checksum is equal to zero, meaning that the found substring represents a subtree in its prefix notation.

5.2 Experiments

The aim of the implementation is to compare and contrast the performance of the implemented solutions to the subtree rejection problem. In order to assess the efficiency of the algorithms, we decided to measure the following data:

- the build time,
- the query time,
- the memory consumption,
- the number of an automaton's states, and
- the number of an automaton's transitions.

The build time, the average query time, and the memory consumption of an indexing algorithm are relevant to all tree indexing solutions. The rest of the data are specific to automata solutions. The latter two are simple integers that represent the number of instances of the two factors. In the rest of this section, we state how precisely we measure the time and memory consumption of the program.

The time data are measured in a separate run from the run analyzing memory because computing the memory consumption of a C++ program negatively affects its time duration. Both data are collected on a program compiled with the `O2` flag. `O2` flag means that the code is optimized as much as possible without a space-speed trade-off [57].

The metric we use to measure the build and query time is the processor time used by these operations, calculated using `std::clock` [58]. We measure the build time by timing how long it takes to complete a call to the `BuildIndex` method for each tree separately. Unlike the build time, we measure the query time by calculating the time it takes to complete all queries on an index made for one tree. Measurement includes the `for` loop to iterate over all queries and to output the results. Each query is run ten times. Since all three indexes are tested under the same conditions, we can use these data to compare their performance.

We measure the memory consumption of the different indexes by the maximum allocated bytes on the heap at one time. To measure the peak memory consumption, we use Valgrind's heap profiler *Massif* [59]. Massif calculates the memory consumption of the whole program; therefore, the peak memory consumption does not reflect only the size of a subtree rejection index but also the memory used to store other data. However, since the memory is determined for each automaton under the same circumstances, we can compare the obtained values.

Chapter 6

Experiment results

In this chapter, we present experiments that compare our implementations of three solutions to the subtree rejection problem. The solutions are: subtree pushdown automata, subtree oracle pushdown automata, and subtree finite automata. To compare and assess the performance of the three algorithms, we measure the following data:

- the CPU build time in milliseconds,
- the CPU query time in milliseconds,
- the peak memory consumption of the program in bytes,
- the number of an automaton's states, and
- the number of an automaton's transitions.

The implementation of the algorithms and the collection of the experiment results are described in Chapter 5. In the rest of this chapter, we present the data on which the three indexes were tested and outline the results of the experiments.

To test the indexing algorithms on a wide range of different sizes and heights of trees and queries, we generated a dataset using the *Algorithms Library Toolkit* [60]. The generated dataset was rewritten into a format where each node is represented by an integer representing the node's rank. Therefore, all trees and queries are over a unique rank alphabet. The highest rank in the dataset is 21; however, most nodes have a much lower rank.

Altogether, there are 520 different trees. The lowest height of a tree is 1, and the highest is 15. For each height, the dataset has sparse as well as dense trees; therefore, 15 was chosen as the maximum height because making dense trees with larger heights would require exponentially more nodes. The lowest number of nodes in a tree is 2, and the highest is 65 660. The number of nodes in a tree varies from around $2^{h+1} - 2^h$ to around 2^{h+1}, where h is the height of a tree.

As for the queries, there are five distinct datasets, each with 100 trees, so that higher trees can have larger query trees. The datasets differ in the maximum height of a tree, and the heights are 1, 2, 3, 4, and 6. The minimum size of a query tree is 1, and the maximum size is 127. The query trees are represented in the same way as the input trees; therefore, they are all over a unique rank alphabet. The maximum rank of a node is 10. The pairings of trees and query sets are as follows:

- trees of height equal to 1 are queried with trees of height up to 1,
- trees of height equal to 2 and 3 are queried with trees of height up to 2,
- trees of height equal to 4 to 7 are queried with trees of height up to 3,

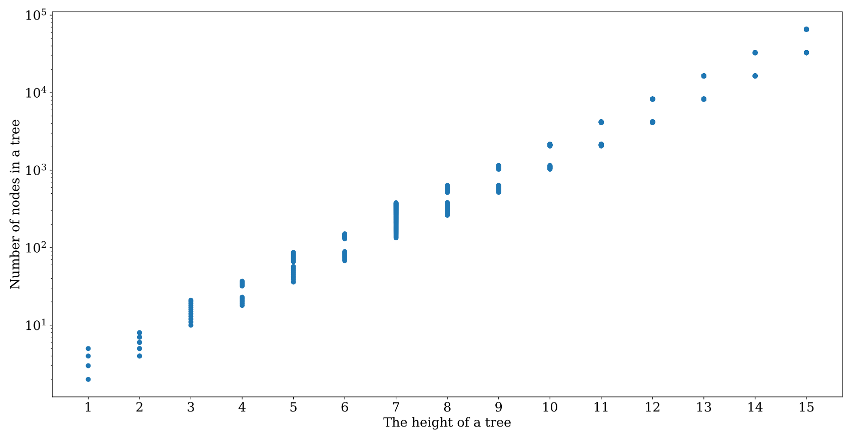


Figure 6.1 A graph showing the distribution of heights and sizes of trees in the input trees dataset.

- trees of height equal to 8 to 11 are queried with trees of height up to 4, and
- trees of height equal to 12 to 15 are queried with trees of height up to 6.

In Figure 6.1, we can see a graph showing the distribution of the number of nodes in a tree and the height of a tree for the input trees dataset. The graph shows that as the trees increase in height, they also have more nodes. Furthermore, we can see that for most of the heights, the trees are divided into two groups, one with a lower number of nodes and one with more nodes.

Across all the implemented automata, subtree finite automata consistently had by far the most states and transitions. The difference in the number of states and transitions of subtree pushdown automata and subtree oracle pushdown automata was not as significant. Subtree oracle pushdown automata have $n + 1$ states, where n is the size of a tree [12]. However, that is the number of states of a nondeterministic automaton; therefore, the difference between deterministic subtree pushdown automata and deterministic subtree oracle pushdown automata was not as large as we first anticipated. Nevertheless, subtree oracle pushdown automata consistently had the least states and transitions. Furthermore, as the input trees got larger, so did the difference in the number of states and transitions between subtree finite automata and the other two automata. To demonstrate, the subtree FA index for the last input tree had 658 557 states, while the subtree PDA had only 72 212 and the subtree oracle PDA 68 220 states.

Figures 6.2 and 6.3 show the relationship between the number of states of an automaton and the height and size of a tree. Both figures indicate that the number of states is primarily dependent on the size of the input tree, not on its height. As we can see in Figure 6.2, the number of states corresponds directly to the number of nodes in a tree. In Figure 6.3, we can see the general trend that the number of states grows, as does the height of a tree. However, from a certain point onward, there is a gap for each height, which suggests that the number of states corresponds more to the size of an input tree than its height. The graphs showing the relationship between the number of transitions and the height and size of a tree are nearly identical to the graphs in Figures 6.2 and 6.3. Thus, we placed them in the Appendix.

It appears that the time it took to complete the queries depended primarily on the size of an input tree for all three solutions. All implemented indexing solutions have their query time equal to $O(m \cdot \log(t))$, where m is the size of a query and t is the number of transitions in a specific automaton. Therefore, since the query time was also affected by the number of transitions in an automaton, it is unsurprising that the query time rose even when the query sets were the same. Thus, we anticipated that subtree finite automata would have the longest query time, as they consistently had by far the most transitions. Nevertheless, they performed by far the best in this regard. The second least time took subtree oracle pushdown automata, and the most time took subtree pushdown automata.

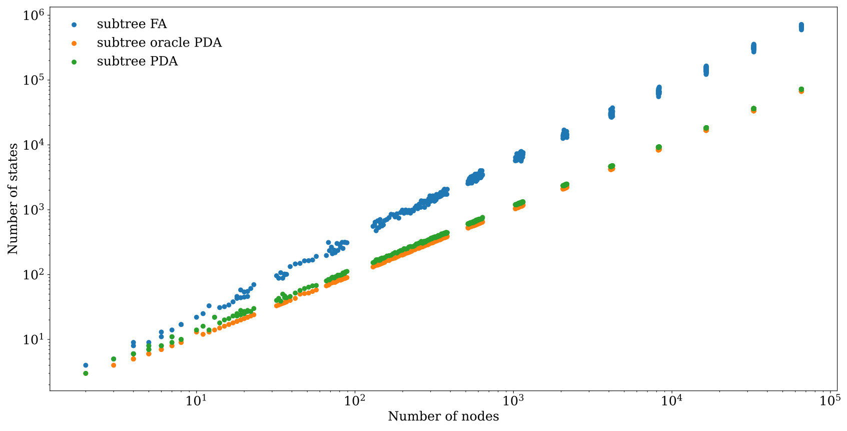


Figure 6.2 The number of states of the three implemented automata given the size of an input tree.

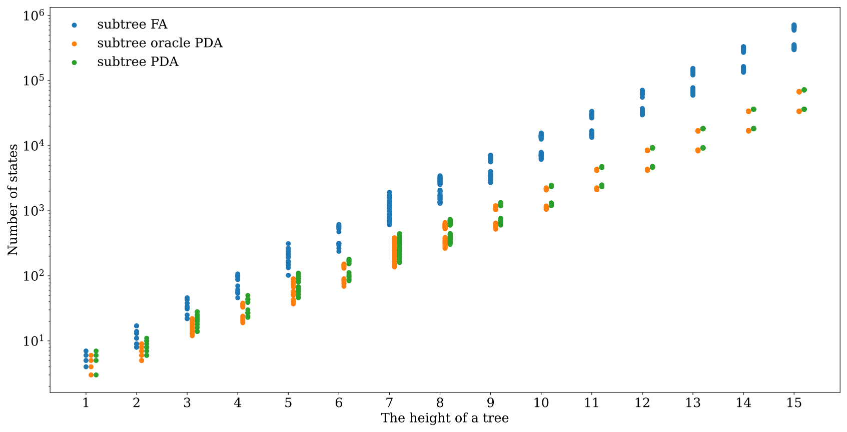


Figure 6.3 The number of states of the three implemented automata given the height of an input tree.

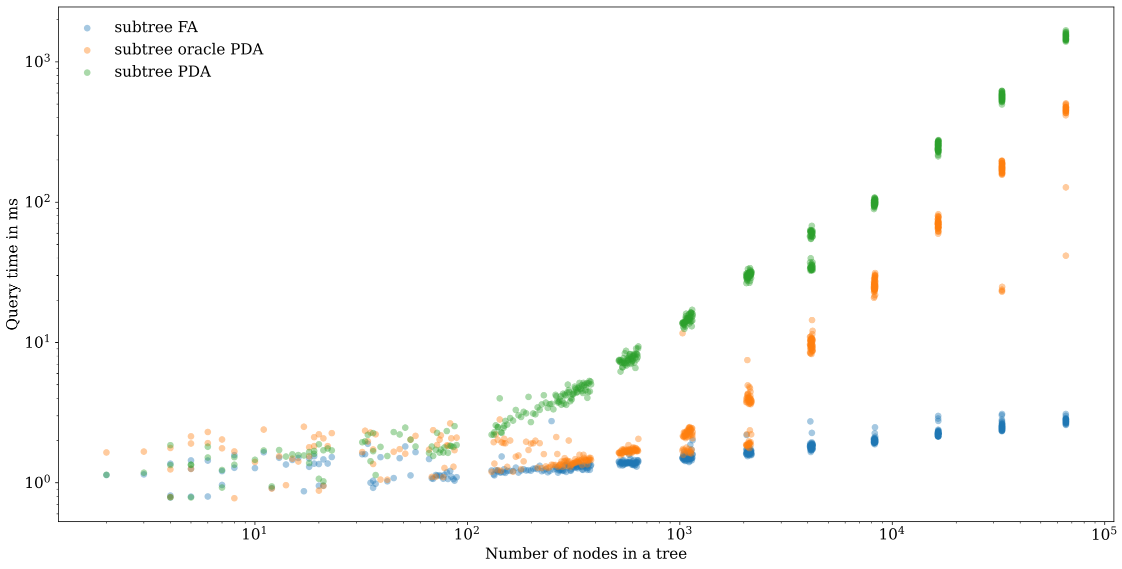


Figure 6.4 The time it took to complete all the queries given the number of nodes in an input tree. We can see that apart from a few outliers, the query time is proportional to the size of an input tree.

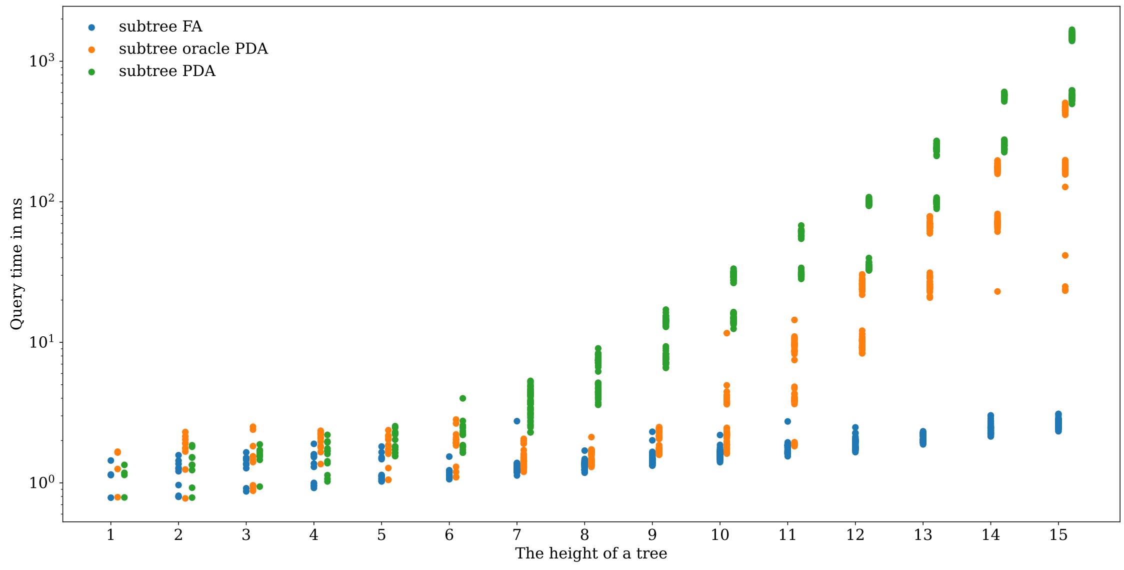


Figure 6.5 The time it took to complete all the queries given the height of an input tree. We believe that for subtree pushdown automata and subtree oracle pushdown automata, the height of a tree does not contribute to the query time as much as the size of a tree. We draw this conclusion from the fact that the query time distribution for the two automata approximately follows the distribution of the number of nodes given a specific height of an input tree, as is seen in Figure 6.3).

We believe that one reason why subtree pushdown automata performed the worst in the query time is that there is inefficiency in naming the states. Although all the implemented automata use the same class to represent an automaton's state, subtree oracle pushdown automata and subtree finite automata always have only one integer in the set that describes the label of a state. In contrast, subtree pushdown automata can have multiple integers in the label set because each of the integers represents a position within the input tree. This might negatively affect the query time because the comparison of any two states takes longer.

However, both subtree oracle pushdown automata and subtree finite automata have label sets of size one, and yet subtree finite automata took less time. We believe that this is because subtree oracle pushdown automata had to compute and work with pushdown stores, while subtree finite automata did not.

In terms of build time, the subtree oracle pushdown automata took the most time, and subtree finite automata the least, as is seen in Figure 6.6. Subtree oracle pushdown automata inherently take longer to build than subtree pushdown automata. The reason is that the process of building a subtree oracle PDA first creates a subtree PDA and then modifies it. Subtree finite automata performed the best in the build time, presumably because there were no further necessary steps once an automaton was created. The algorithm we used created a deterministic finite automaton right away; therefore, no determinization or removal of inaccessible states was needed, in contrast to pushdown automata solutions.

Unfortunately, the information about the peak memory consumption of the programs, which we can see in Figure 6.7, is not as relevant as we had hoped. The primary reason why we question the relevancy of the data is that subtree oracle pushdown automata performed the worst of the three solutions despite the fact that subtree pushdown automata had overall more states, transitions, and the labels of its states can have multiple integers, which we expected to be reflected in the peak memory consumption information. Moreover, subtree finite automata performed the best even though they have many more states than the other two pushdown automata indexes. Both of those examples show that the peak memory consumption does not reflect only the size of the resulting index but also the memory consumption of creating the

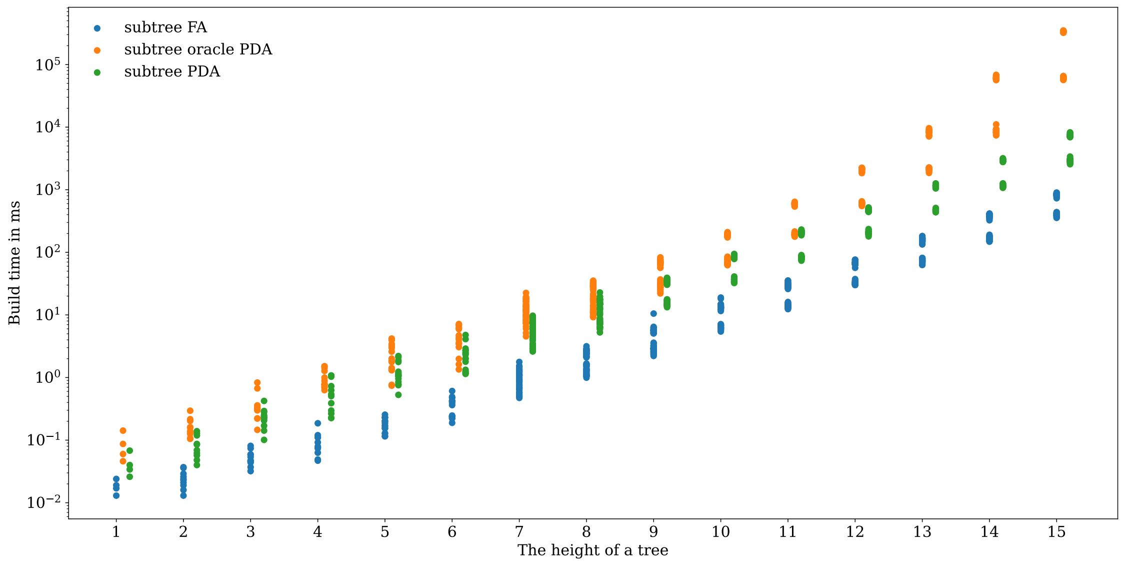


Figure 6.6 The time it took to build an index given the height of an input tree.

index. Therefore, unless we have constrained memory space when building the index, the data indicating the peak memory consumption have to be taken with some reservations.

These experiments provided us with some basic insight into how well the indexes perform in practice. Of the three indexes that we tested, subtree finite automata performed best with regard to query time, build time, and even peak memory consumption; however, peak memory consumption does reflect only the size of the indexing structure; therefore, it has to be taken with some reservations. For future work, it would be useful to investigate further how the three indexes perform under these scenarios:

- on a wider range of sizes of trees of the same height,
- on ranked trees over a non-unique rank alphabet,
- on trees in which we could affect the frequency of certain subtrees,
- on tests specifically designed to investigate how common are false positives in subtree oracle pushdown automata answers,
- on subtree pushdown automata that use only one integer as a name of a state instead of a set of integers, and
- to test the size of the representation of a given automaton (not the peak memory consumption of the whole program).

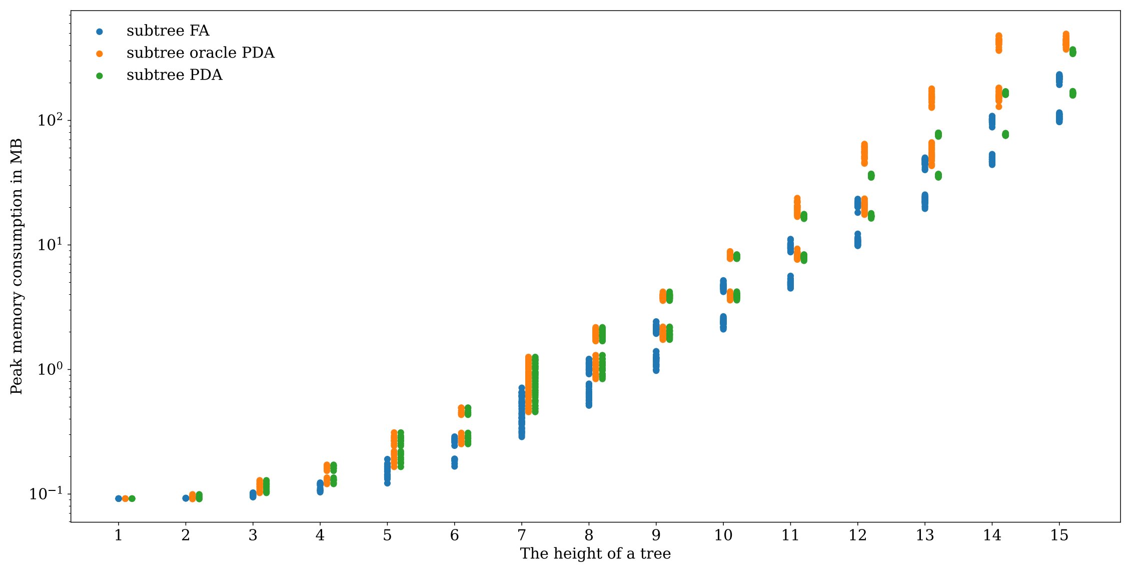


Figure 6.7 The peak memory consumption of a program running an automaton index for a tree with a given height.

Chapter 7

Conclusions

In this chapter, we turn to the aims of this thesis formulated in Chapter 1 and examine their fulfillment. We also present suggestions for future work to extend and improve this survey.

The first objective was to survey existing tree indexing problems and definitions used in the literature. We summarized the results of our research in Chapter 3, where we presented four categories that help define a tree indexing problem. The categories are as follows:

- the type of an input tree,
- the type of queries,
- what is considered a match, and
- the type of required answers.

The second aim was to compare and contrast two solutions to a selected tree indexing problem. Within the survey, we presented a definition of the subtree rejection problem to which we later showed viable solutions: subtree PDA [14] and subtree oracle PDA [12], and we also proposed a novel index called subtree FA. All three solutions have a query time equal to $O(m)$, where m is the size of a query, regardless of the size of an input tree. The deterministic subtree PDA has fewer than $2n + 1$ states and at most $3n$ transitions, where n is the size of an input tree. The subtree oracle PDA always has $n + 1$ states. The subtree FA has $O(n^2)$ states as well as transitions.

The third aim was to implement and compare the performance of the three algorithms. Therefore, we conducted an experiment that measured the build time, query time, peak memory consumption, number of states, and number of transitions. Subtree finite automata performed the best in build time, query time, and peak memory consumption. However, the peak memory consumption also reflects the memory consumption of building the indexing structure, not only the size of the structure itself.

In conclusion, all the objectives were achieved. Furthermore, there are many topics for future research that extend the presented results:

- oracle modification of subtree finite automata for better space efficiency with a comparison to subtree oracle pushdown automata,
- survey on parallel solutions to tree indexing problems,
- approximate tree indexing using finite automata, and
- survey on problems building on tree indexing, such as subtree mining and finding frequent subtrees.

Experiment results in graphs

In this chapter, we place graphs showing the results of our experiments that were not included in the main part of this thesis.

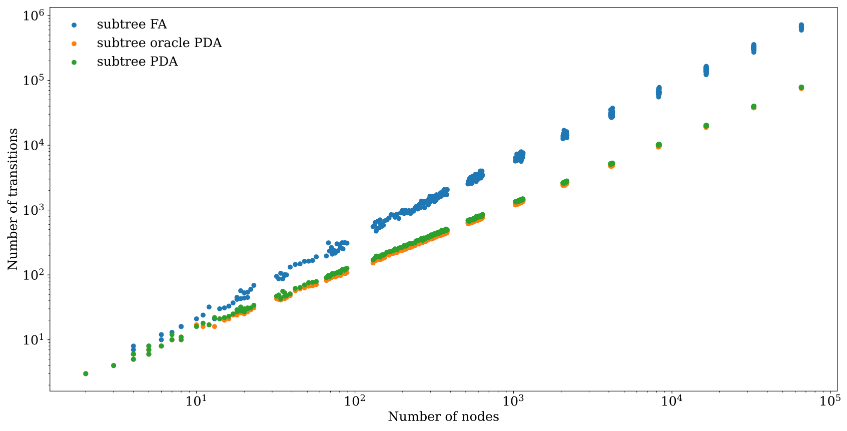


Figure A.1 The number of transitions of the three implemented automata given the size of an input tree.

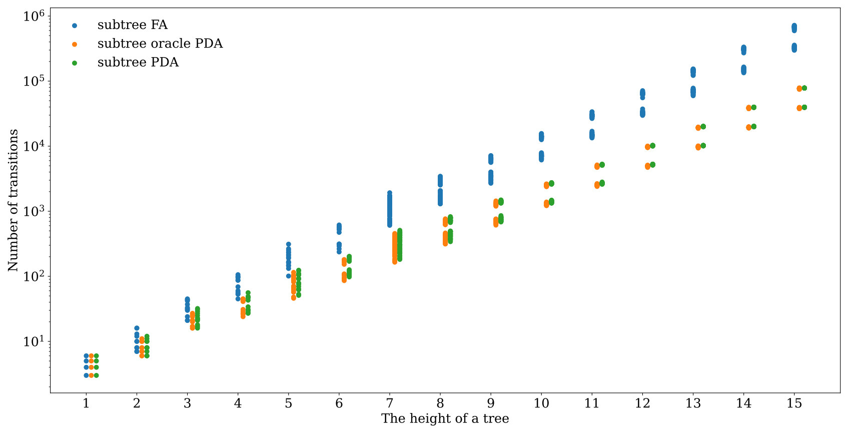


Figure A.2 The number of transitions of the three implemented automata given the height of an input tree.

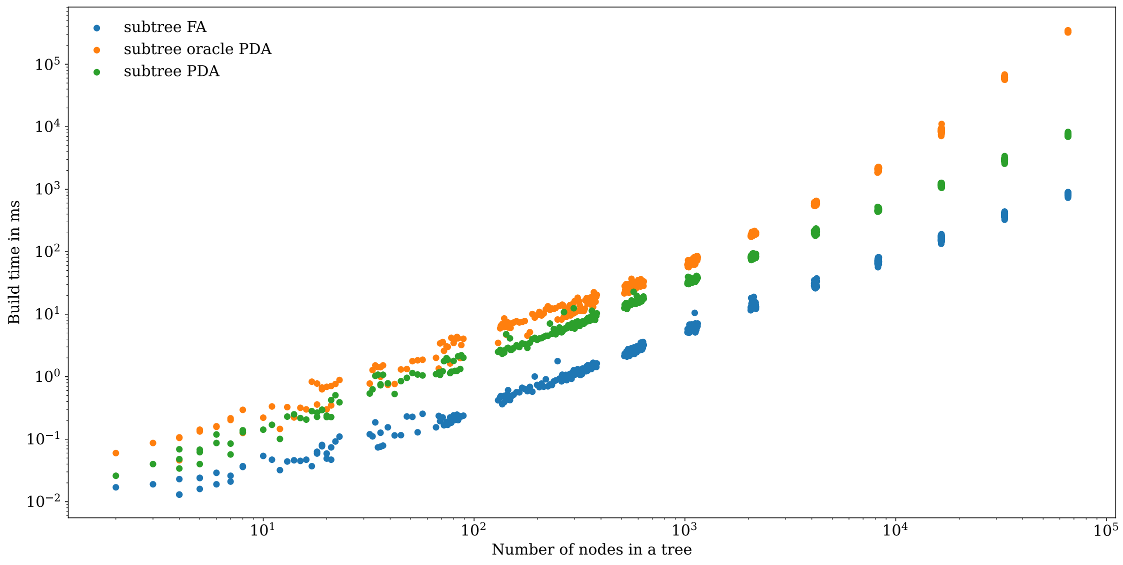


Figure A.3 The time it took to build an index given the size of an input tree.

Bibliography

1. CHI, Y.; YANG, Y.; MUNTZ, R. R. Indexing and mining free trees. In: *Third IEEE International Conference on Data Mining*. 2003, pp. 509–512. Available from DOI: `10.1109/ICDM.2003.1250964`.
2. FLOURI, T.; ILIOPOULOS, C.; JANOUŠEK, J.; PISSIS, S. P. Tree indexing by pushdown automata and subtree repeats. In: *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*. 2011, pp. 899–902. Available also from: `http://dx.doi.org/`.
3. ASAI, T.; ARIMURA, H.; UNO, T.; NAKANO, S.-i. Discovering Frequent Substructures in Large Unordered Trees. In: *6th International Conference on Discovery Science*. Springer Verlag, 2003. ISSN 0302-9743. Available from DOI: `10.1007/978-3-540-39644-4\_6`.
4. JANOUŠEK, J. *Arbology: Algorithms on Trees and Pushdown Automata*. 2010. Vysoké učení technické v Brně.
5. MELICHAR, B.; HOLUB, J.; POLCAR, T. Text Searching Algorithms-Volume I: Forward String Matching. *Department of Computer Science and Engineering, Czech Technical University*. 2005.
6. ŠESTÁKOVÁ, E. *Indexing XML Documents and Tree Data Structures* [A Doctoral Study Report]. 2017.
7. CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms, third edition*. MIT Press, 2009. ISBN 9780262258104.
8. CLEOPHAS, L. *Tree algorithms: two taxonomies and a toolkit*. Technische Universiteit Eindhoven, 2008. ISBN 9789038612287. Available from DOI: `10.6100/IR633481`.
9. HOPCROFT, J. E.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. ISBN 9780201029888.
10. TRAVNÍČEK, J.; JANOUŠEK, J.; MELICHAR, B. Indexing ordered trees for (nonlinear) tree pattern matching by pushdown automata. *Computer Science and Information Systems*. 2012, vol. 9, no. 3, pp. 1125–1153. Available also from: `http://www.doiserbia.nb.rs/Article.aspx?id=1820-02141200024T`.
11. TRÁVNÍČEK, J.; JANOUŠEK, J.; MELICHAR, B. Indexing trees by pushdown automata for nonlinear tree pattern matching. In: *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2011, pp. 871–878. Available also from: `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6078238`.
12. PLICKA, M.; JANOUŠEK, J.; MELICHAR, B. Subtree Oracle Pushdown Automata for Ranked and Unranked Ordered Trees. In: 2011. ISBN 9788360810392.

13. POLIAK, M. *On Indexes of Ordered Trees for Subtrees and Tree Patterns and Their Space Complexities.* 2017. PhD thesis. Faculty of Information Technology, Czech Technical University in Prague,

14. JANOUŠEK, J. String Suffix Automata and Subtree Pushdown Automata . *Proceedings of the Prague Stringology Conference 2009.* 2009.

15. COHEN, S. *Indexing for subtree similarity-search using edit distance.* 2013. Available from DOI: `10.1145/2463676.2463716`.

16. FERRAGINA, P.; LUCCIO, F.; MANZINI, G.; MUTHUKRISHNAN, S. Compressing and indexing labeled trees, with applications. *J. ACM.* 2009, vol. 57, no. 1, pp. 1–33. ISSN 0004-5411. Available from DOI: `10.1145/1613676.1613680`.

17. JANOUŠEK, J.; MELICHAR, B.; POLÁCH, R.; POLIAK, M.; TRÁVNÍČEK, J. *A Full and Linear Index of a Tree for Tree Patterns.* 2014. Available from DOI: `10.1007/978-3-319-09704-6\_18`.

18. WANG, H.; PARK, S.; FAN, W.; YU, P. S.; THOMAS, I.; WATSON, J.; CENTER, R. *ViST: A dynamic index method for querying XML data by tree structures.* 2003. Available also from: `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.6517&rep=rep1&type=pdf`. Accessed: 2022-01-01.

19. CATANIA, B.; MADDALENA, A.; VAKALI, A. XML document indexes: a classification. *IEEE Internet Comput.* 2005, vol. 9, no. 5, pp. 64–71. ISSN 1089-7801, ISSN 1941-0131. Available from DOI: `10.1109/MIC.2005.115`.

20. LUK, R. W. P.; LEONG, H. V.; DILLON, T. S.; CHAN, A. T. S.; CROFT, W. B.; ALLAN, J. A survey in indexing and searching XML documents. *J. Am. Soc. Inf. Sci. Technol.* 2002, vol. 53, no. 6, pp. 415–437. ISSN 1532-2882, ISSN 1532-2890. Available from DOI: `10.1002/asi.10056`.

21. RAO, P. Indexing XML data for efficient twig pattern matching. 2007. Available also from: `https://repository.arizona.edu/handle/10150/194425`.

22. DO THANH TUNG, H.; DUC LUONG, D. An improved indexing method for xpath queries. *Indian J. Sci. Technol.* 2016, vol. 9, no. 31. ISSN 0974-6846, ISSN 0974-5645. Available from DOI: `10.17485/ijst/2016/v9i31/92731`.

23. CHUNG, C.-W.; MIN, J.-K.; SHIM, K. *APEX: An adaptive path index for XML data.* 2002. Available also from: `https://resources.mpi-inf.mpg.de/d5/teaching/ss03/xml-seminar/Material/chung_sigmod2002.pdf`. Accessed: 2021-12-19.

24. ŠESTÁKOVÁ, E.; JANOUŠEK, J. Automata Approach to XML Data Indexing. *Information.* 2018, vol. 9, no. 1, p. 12. ISSN 1343-4500. Available from DOI: `10.3390/info9010012`.

25. MANDHANI, B.; SUCIU, D. Query caching and view selection for XML databases. In: *Proceedings of the 31st international conference on Very large data bases.* Trondheim, Norway: VLDB Endowment, 2005, pp. 469–480. VLDB '05. ISBN 9781595931542. Available also from: `https://dl.acm.org/doi/pdf/10.5555/1083592.1083648`.

26. MAHBOUBI, H.; DARMONT, J. Indices in XML Databases. In: *Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends.* IGI Global, 2009, pp. 674–681. Available from DOI: `10.4018/978-1-60566-242-8.ch072`.

27. MENG, X.; JIANG, Y.; CHEN, Y.; WANG, H. XSeq: an indexing infrastructure for tree pattern queries. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data.* Paris, France: Association for Computing Machinery, 2004, pp. 941–942. SIGMOD '04. ISBN 9781581138597. Available from DOI: `10.1145/1007568.1007709`.

28. PRAVEEN RAO; MOON, B. PRIX: indexing and querying XML using prufer sequences. In: *Proceedings. 20th International Conference on Data Engineering.* ieeexplore.ieee.org, 2004, pp. 288–299. ISSN 1063-6382. Available from DOI: `10.1109/ICDE.2004.1320005`.

29. ZOU, Q.; LIU, S.; CHU, W. W. Ctree: a compact tree for indexing XML data. In: *Proceedings of the 6th annual ACM international workshop on Web information and data management*. Washington DC, USA: Association for Computing Machinery, 2004, pp. 39–46. WIDM '04. ISBN 9781581139785. Available from DOI: 10.1145/1031453.1031462.

30. JIANG, H.; LU, H.; WANG, W.; OOI, B. C. XR-tree: indexing XML data for efficient structural joins. In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 2003, pp. 253–264. Available from DOI: 10.1109/ICDE.2003.1260797.

31. LI, Q.; MOON, B. Indexing and querying XML data for regular path expressions. In: *Proceedings of the 27th VLDB Conference*. 2001. Available also from: http://www.vldb.org/conf/2001/P361.pdf.

32. KIM, S.-H.; CHO, H.-G. A compact index for Cartesian tree matching. In: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. Available from DOI: 10.4230/LIPICS.CPM.2021.18.

33. PARK, S. G.; AMIR, A.; LANDAU, G. M.; PARK, K. Cartesian Tree Matching and Indexing. 2019. Available from DOI: 10.4230/LIPIcs.CPM.2019.13.

34. ŽĎÁREK, J.; MELICHAR, B. A Note on a Tree-Based 2D Indexing. In: *Implementation and Application of Automata*. Springer Berlin Heidelberg, 2011, pp. 300–309. Available from DOI: 10.1007/978-3-642-18098-9_32.

35. ŽĎÁREK, J.; MELICHAR, B. TREE-BASED 2D INDEXING. *Internat. J. Found. Comput. Sci.* 2011, vol. 22, no. 08, pp. 1893–1907. ISSN 0129-0541. Available from DOI: 10.1142/S0129054111009100.

36. MILO, T.; SUCIU, D. Index Structures for Path Expressions. In: *Database Theory — ICDT'99*. Springer Berlin Heidelberg, 1999, pp. 277–295. Available from DOI: 10.1007/3-540-49257-7_18.

37. ZHOU, G. Path Queries in Weighted Trees. 2012. Available also from: https://core.ac.uk/display/144145816?recSetID=.

38. JIEFENG CHENG; GE YU; GUOREN WANG; YU, J. X. PathGuide: an efficient clustering based indexing method for XML path expressions. In: *Eighth International Conference on Database Systems for Advanced Applications, 2003. (DASFAA 2003). Proceedings.* ieeexplore.ieee.org, 2003, pp. 257–264. Available from DOI: 10.1109/DASFAA.2003.1192390.

39. TRÁVNÍČEK, J. *(Nonlinear) Tree Pattern Indexing and Backward Matching*. 2018. PhD thesis. Faculty of Information Technology, Czech Technical University in Prague.

40. BILLE, P.; GØRTZ, I. L.; VILDHØJ, H. W.; WIND, D. K. String matching with variable length gaps. *Theor. Comput. Sci.* 2012, vol. 443, pp. 25–34. ISSN 0304-3975. Available from DOI: 10.1016/j.tcs.2012.03.029.

41. LAOHAWEE, P.; TANGPONG, A.; RUNGSAWANG, A. Parallel DSIR Text Indexing System: Using Multiple Master/Slave Concept. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, 2000, pp. 297–303. Available from DOI: 10.1007/3-540-45255-9_41.

42. FERRAGINA, P. Dynamic Text Indexing under String Updates. *J. Algorithm. Comput. Technol.* 1997, vol. 22, no. 2, pp. 296–328. ISSN 1748-3018, ISSN 0196-6774. Available from DOI: 10.1006/jagm.1996.0814.

43. NARANG, A.; AGARWAL, V.; KEDIA, M.; GARG, V. K. Highly scalable algorithm for distributed real-time text indexing. In: *2009 International Conference on High Performance Computing (HiPC)*. 2009, pp. 332–341. ISSN 1094-7256. Available from DOI: 10.1109/HIPC.2009.5433193.

44. GHOTING, A.; MAKARYCHEV, K. I/O efficient algorithms for serial and parallel suffix tree construction. *ACM Trans. Database Syst.* 2010, vol. 35, no. 4, pp. 1–37. ISSN 0362-5915. Available from DOI: `10.1145/1862919.1862922`.

45. CHRISTIANSEN, A. R.; FARACH-COLTON, M. Parallel Lookups in String Indexes. In: *String Processing and Information Retrieval*. Springer International Publishing, 2016, pp. 61–67. Available from DOI: `10.1007/978-3-319-46049-9\_6`.

46. GHOTING, A.; MAKARYCHEV, K. Serial and parallel methods for i/o efficient suffix tree construction. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 827–840. SIGMOD '09. ISBN 9781605585512. Available from DOI: `10.1145/1559845.1559931`.

47. LI, Q. Indexing and path query processing for XML data. 2004. Available also from: `https://repository.arizona.edu/handle/10150/290141`.

48. ATHANASSOULIS, M.; AILAMAKI, A. BF-tree: approximate tree indexing. In: *Proceedings of the 40th International Conference on Very Large Databases*. infoscience.epfl.ch, 2014. Available also from: `https://infoscience.epfl.ch/record/201942`.

49. JAWARI KAPISHA, S.; VIJAYA LAKSHMI, G. Exploring XML Index Structures and Evaluating C-Tree Index-based Algorithm. In: *2020 3rd International Conference on Intelligent Sustainable Systems (ICISS)*. 2020, pp. 212–218. Available from DOI: `10.1109/ICISS49785.2020.9316052`.

50. TANG, N.; YU, J. X.; OZSU, M. T.; WONG, K. Hierarchical Indexing Approach to Support XPath Queries. In: *2008 IEEE 24th International Conference on Data Engineering*. ieeexplore.ieee.org, 2008, pp. 1510–1512. ISSN 2375-026X. Available from DOI: `10.1109/ICDE.2008.4497606`.

51. CROCHEMORE, M.; ILIE, L.; SEID-HILMI, E. The structure of Factor Oracles. *World Scientific Publishing House Ltd.* 2007. Available also from: `https://core.ac.uk/display/48347230?recSetID=`.

52. MANCHERON, A.; MOAN, C. Combinatorial Characterization of the Language Recognized by Factor and Suffix Oracles. *World Scientific Pub Co Pte Lt.* 2005. Available also from: `https://core.ac.uk/display/53013579?recSetID=`.

53. ALLAUZEN, C.; CROCHEMORE, M.; RAFFINOT, M. Factor oracle : a new structure for pattern matching. *Springer Science and Business Media LLC.* 1999. Available also from: `https://core.ac.uk/display/48347101?recSetID=`.

54. CLEOPHAS, L.; KOURIE, D. G.; WATSON, B. W. Weak factor automata : the failure of failure factor oracles? *South African Institute of Computer Scientists and Information Technologists.* 2014. Available also from: `https://core.ac.uk/display/188222317?recSetID=`.

55. *std::map*. [N.d.]. Available also from: `https://en.cppreference.com/w/cpp/container/map`. Accessed: 2022-03-30.

56. *std::set*. [N.d.]. Available also from: `https://en.cppreference.com/w/cpp/container/set`. Accessed: 2022-03-30.

57. *Optimize Options (Using the GNU Compiler Collection (GCC))*. [N.d.]. Available also from: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`. Accessed: 2022-03-30.

58. *std::clock*. [N.d.]. Available also from: `https://en.cppreference.com/w/cpp/chrono/c/clock`. Accessed: 2022-03-24.

59. THE VALGRIND DEVELOPERS. *Valgrind*. [N.d.]. Available also from: `https://valgrind.org/docs/manual/ms-manual.html`. Accessed: 2022-03-24.

60. TRÁVNÍČEK, J.; PECKA, T.; ŠTĚPÁN, P., et al. *Algorithms Library Toolkit*. [N.d.]. Available also from: `https://alt.fit.cvut.cz/`.

Contents of enclosed media

The project is also available at `https://gitlab.fit.cvut.cz/hrncikar/bap-implementation`.

```
README.md.......................................the file with media contents description
├── experimentsMemory.sh............................the script to test memory consumption
├── experimentsTime.sh....................................the script to test execution time
├── data............................the directory of generated datasets and experiment results
├── src .......................................... the directory of implementation sources
└── text
    ├── thesis ................................the directory of LaTeX source codes of the thesis
    └── thesis.pdf ......................................... the thesis text in PDF format
```