Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

On Indexes of Ordered Trees for Subtrees and Tree Patterns and Their Space Complexities

by

Ing. Martin Poliak

A thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

PhD programme: Informatics

Prague, June 2017

Thesis Supervisor:

Doc. Ing. Jan Janoušek, Ph.D.
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Abstract

This doctoral thesis deals with methods of indexing of a tree for subtrees and for tree patterns. Two types of indexes are considered. The first type is the index of a tree for subtrees, i.e. a full index that accepts all subtrees of a given tree. The second type is the index of a tree for tree patterns, i.e. a full index that accepts all tree patterns that match a given tree at any of its nodes. The results of the doctoral thesis are divided into three parts.

As the first result, this doctoral thesis presents a deterministic pushdown automaton called *tree compression automaton* (TCA), which can be used for multiple purposes. Firstly, as an index of the subject tree(s) for subtrees. Secondly, as a subtree matcher. Thirdly, TCA can be used for computing subtree repeats. Lastly, it can be used for compression of indexed tree(s). A conversion algorithm from a TCA to a finite tree automaton (FTA) [18] is given.

As the second result, this doctoral thesis presents a linear-space index of a tree for tree patterns. A fast searching algorithm that uses this index is given. It is shown that the presented index, together with the searching algorithm, is an efficient simulation of a non-deterministic tree pattern pushdown automaton, which accepts all tree patterns that match a given tree.

As the third result, this doctoral thesis investigates the space complexities of deterministic finite tree automata and deterministic tree pattern pushdown automata. Both automata that represent an index of a tree for tree patterns and they have non-deterministic variants with linear size. This text shows that there exist trees such that any deterministic finite tree automaton used as an index of these trees for tree patterns has size exponential with respect to the indexed trees. A related result is demonstrated for deterministic tree pattern PDAs.

The results are a part of arbology research [50]. Arbology is an algorithmic discipline dealing with processing of trees that bases its approach on pushdown automata.

Tree compression automaton

Tree compression automaton (TCA) is a specific deterministic pushdown automaton that is shown to be suitable for indexing of a tree for subtrees, for subtree matching, locating subtree repeats and for tree compression. The TCA accepts by empty pushdown store all subtrees in prefix bar notation [3] of trees in a given set of trees T.

An on-line and incremental construction algorithm for TCA is presented. The construction algorithm creates a TCA whose size is in the worst case linear with respect to the size of the indexed tree(s). In the best case, the size of the created TCA is logarithmic. This property of TCA can be used for compression of the indexed tree(s).

A TCA for a tree with n nodes has at most $n+1$ states, $2n+1$ pushdown store symbols and the number of transition rules is $4n$. If a hash map is used for the storage of the transition function of a TCA, the construction of a TCA for a tree with n nodes takes time $\mathcal{O}(2n)$ and requires working space of size at most $2n$.

A linear-time decompression algorithm for TCA is presented. The compression and decompression performance of TCA is verified experimentally and compared to other compression methods. A library that provides compression by TCA is available [52].

An algorithm for subtree matching that uses TCA is introduced. Given a tree t with n nodes and a set of trees T, the algorithm reports all subtrees of tree t that match trees in set T in time $\mathcal{O}(2n)$ if hashing is used.

An algorithm for finding exact repeats of subtrees in a set of trees is presented. The algorithm for finding exact repeats takes linear time with respect to the size of the input when a hash map is used for the storage of transition function δ.

The tree compression automaton is put into context of finite tree automata (FTA). A conversion algorithm from a TCA into a deterministic FTA that accepts the same trees is given.

A full and linear index of a tree for tree patterns

For indexing of a tree for tree patterns, arbology presents a tree pattern pushdown automaton (tree pattern PDA). This automaton can have size exponential with respect to the size of the indexed tree. This motivated the creation of a linear size index whose usage can be seen as a simulation of a tree pattern PDA. The index consists of a compact suffix automaton [20], and a *subtree jump table*.

Given a subject tree t with n nodes, the indexing phase is proved to take $\mathcal{O}(n)$ time and require $\mathcal{O}(n)$ space. The number of distinct tree patterns which match the tree is $\mathcal{O}(2^n)$, but the index that is built during the indexing phase requires only $\mathcal{O}(n)$ space.

The searching phase reads an input tree pattern P of size m and locates all its occurrences in tree t. For an input tree pattern P in linear prefix notation $pref(P) = P_1SP_2S\dots SP_k$, $k \geq 1$, the searching is performed in time $\mathcal{O}(m + \sum_{i=1}^{k} |occ(P_i)|)$, where $occ(P_i)$ is the set of all occurrences of string P_i in $pref(t)$.

On space requirements of indexes based on FTA and on tree pattern PDA

A specific deterministic finite tree automaton suitable for indexing of a tree for tree patterns is presented. Trees are shown such that all deterministic FTAs that index them have exponential size with respect to the size of the indexed trees, whereas the non-deterministic FTAs that index them have linear size.

More precisely, let t be a tree over alphabet $\mathcal{A}$. Let N_t^S denote a non-deterministic finite tree automaton (NFTA) that accepts all tree patterns that match t. Let D_t^S denote deterministic FTA (DFTA) equivalent to N_t^S. Then there exists a tree td with n nodes (shown in the text of the doctoral thesis), whose NFTA N_{td}^S has $\mathcal{O}(n)$ states and whose minimum complete DFTA D_{td}^S has $\mathcal{O}(2^{n/4})$ states.

An analogous result is shown for tree pattern PDAs. It is shown that there exists a tree

t of size n such that the deterministic tree pattern PDA that indexes it has exponential size $\mathcal{O}(2^{n/4})$. This proof is an improvement of a related result from arbology [29].

Keywords:
Tree Indexing, Tree Pattern Indexing, Subtree Matching, Tree Compression, Tree Repeats, Tree Compression Automaton, Finite Tree Automaton, Tree Indexing Complexity

Acknowledgements

I would like to express my gratitude to my doctoral thesis supervisor, Jan Janoušek, Ph.D., associate professor of computer science. He has been a constant source of encouragement and insight during my research and helped me with numerous problems and professional advancements. I valued especially his guidance in my research and his experience through which he could identify and open the interesting and promising topics that eventually lead to this thesis. Praise should also be given to Jan Janoušek's support and proofreading during the writing of this thesis.

I would like to thank professor Bořivoj Melichar for his guidance in the area of stringology and arbology. His years of experience and valuable insight into the field of theoretical informatics together with his focus on detail and formal correctness have been a tremendous help. The captivating theoretic informatics courses lead by Jan Janoušek and Bořivoj Melichar during my master's program motivated me to continue as a Ph.D. student.

Many thanks go to the members of the Department of Theoretical Computer Science, especially professor Jan Holub, Ph.D., Jan Trávníček, Radomír Polách and Martin Šlapák, who helped me with my research and who maintained a pleasant and flexible environment.

I would like to express special thanks to my thesis supervisor, Jan Janoušek, and the department management for providing most of the funding for my research.

Finally, my greatest thanks go to my family members, for their unending patience and care. To my wife Lenka for her loving support, to my mother Mahulena for her support and encouragement throughout my years at the university and to all my beloved ones for their help during my studies.

Dedication

Lence, Tomáškovi a Terezce

Contents

List of Figures

List of Tables

Abbreviations

Alphabet, Language, String

$\mathcal{A}$	Alphabet
a, b	Letter of an alphabet
L	Language
ε	Empty string
$\mathcal{A}^*$	Set of all strings over $\mathcal{A}$, including ε
$\mathcal{A}^+$	$\mathcal{A}^* \setminus \{\varepsilon\}$
w, x, y, z	String

Tree

N	Set of nodes
R	Set of edges
t	Tree
T	Set of trees
r	Root node
$pref(t)$	Prefix notation of tree t
$pref\_par(t)$	Prefix parenthesis notation of tree t
$pref\_bar(t)$	Prefix bar notation of tree t
S	Placeholder for any subtree
p	Tree pattern
occ	Occurrence of a tree pattern

Grammar

G	Grammar
N	Set of non-terminal symbols
P	Set of rules
S	Start symbol of a grammar
$\Rightarrow$	Derivation

Pushdown Automaton

PDA, M	Pushdown automaton
Q	Set of states
G	Pushdown store alphabet
α, β, γ	Strings over $\mathcal{A} \cup G$
δ	Transition function
$\vdash_M$	Transition of automaton M
p, q, r	States
Z, Z_0	Elements of pushdown store alphabet

Finite Tree Automaton

FTA, M	Finite tree automaton
$DFTA$	Deterministic finite tree automaton
$NFTA$	Non-deterministic finite tree automaton
$\mathcal{X}$	Set of variables
C	Tree context
$\underset{M}{\rightarrow}$	Move relation of tree automaton M

Tree Compression Automaton

TCA, M, N	Tree compression automaton
$GTCA$	General tree compression automaton
I	Set of subtree identifiers
μ	Labeling function that assigns subtree identifiers to trees

Other symbols

$M_p(t)$	PDA accepting $pref(t)$
$M_{nps}(t)$	Non-deterministic subtree PDA
$M_{dps}(t)$	Deterministic subtree PDA
$srms(t)$	Set of subtree rightmost states of tree t
$M_{pt}(t)$	Treetop PDA for tree t
$M_{npt}(t)$	Non-deterministic tree pattern PDA for tree t
$M_{dpt}(t)$	Deterministic tree pattern PDA for tree t
$SJT(t)$	Subtree jump table for tree t
$Mc(w)$	Compact suffix automaton for string w
$TPP(p)$	Tree pattern prefix of tree pattern p
p, p_i	Tree pattern, sub-pattern
$occ^t(p_i)$	Set of all occurrences of sub-pattern p_i in $pref(t)$
ac	Arity checksum
m, n, k	Length, number of elements

$\|A\|$	Number of elements in set A
$O(x)$	Big O notation
$\Theta(x)$	Big Θ notation
w.r.t.	Abbreviation for *with respect to*

Chapter 1

Introduction

1.1 Motivation

Trees are one of the key structures in Informatics with their usage ranging from data storage in XML to intermediate representation of programs in compilers. Various kinds of theoretical approaches to analysis and operations on trees exist. Results of this doctoral thesis builds upon the results of the theory of *tree languages and tree automata* [18] and of *arbology* [3].

A crucial property of trees is that they can be easily manipulated when linearised to a string form [50, 3, 38]. The structures and algorithms presented in this doctoral thesis are built upon this property. Algorithmic problems on linearised trees are similar to problems on strings. The input domain of such problems is restricted to linearised forms of trees, which are all strings. However, not all strings are linearised forms of trees. Certain structural constraints can thus be placed on the outputs.

For instance, tree indexing that uses linearized forms of trees typically requires a linearised tree from which an index is built during the indexing phase. During the searching phase, a linearised subtree which is to be located in the indexed tree is required. Compare this to string indexing, which requires a string and a substring for the indexing phase and the searching phase, respectively. The input domain restriction by the algorithmic problems on linearised trees adds a new level of complexity to these problems, but on the other hand offers possibilities for better performance of the algorithms that solve them.

This similarity between the problems on trees and the problems on strings means that the existing algorithms on strings can often be adapted to trees. An existing algorithm on strings for variable length gap matching [7] was an inspiration to an algorithm for building and using a full and linear index of a tree for tree patterns presented in Chapter 5.

1.2 Problem statement

Arbology presents a systematic approach to the study of tree algorithms with the use of pushdown automata [3, 38]. Some of the key areas for which a pushdown automaton was found to be well suited are tree indexing [32], subtree and tree pattern matching [29] or computing repeats in trees [31]. However, pushdown automata presented by arbology always have at least linear size and are thus not suitable for compression of the indexed trees. It would be helpful if the power of pushdown automata used in arbology could be combined with an efficient usage of space.

A regular (recognizable) tree language is a set of trees accepted by some finite tree automaton (FTA). Regular tree languages have similar properties to regular word languages. For instance, regular tree languages are closed under union, intersection and complementation [18]. The set of tree patterns that match a tree is a finite tree language. This implies that there exists a (deterministic) finite tree automaton that can be used as an index of a tree for tree patterns. We have not found any relevant analyses of such automaton, nor has it been shown in the literature what the size of such finite tree automaton is. It is known that tree pattern pushdown automaton can have at worst exponential size with respect to the indexed tree [29]. A hitherto open question asks whether the finite tree automata that accept the set of tree patterns that match a tree also have exponential size.

This work shows that deterministic automata based on the above mentioned approaches (arbology or FTAs) are not suitable as indexes of trees for tree patterns because they have a worst-case exponential size. The non-deterministic versions of the same automata however have linear size. It would thus be helpful if an efficient way to simulate the non-deterministic indexing automata was found.

1.2.1 Tree compression automaton

Indexing of a tree for subtrees can be solved using a *subtree pushdown automaton* [50]. The size of the subtree pushdown automaton depends linearly on the size of the indexed tree. But that is inefficient; trees often contain regularities that can be compressed. Moreover, there are classes of trees (for instance, Fibonacci trees or full n-ary trees) that can be described by a structure logarithmic in size with respect to the size of the tree.

This work introduces a particular pushdown automaton structure called *tree compression automaton* (TCA), which serves as a compressed index of a set of trees. Several theorems are proved about TCA. Its properties are studied - determinism, indexing and matching capabilities, suitability for computing tree repeats. The performance of TCA for compression of trees is examined on real world data. The proposed algorithms based on TCA are on-line and their input are labelled ordered unranked trees.

An on-line and incremental construction algorithm for TCA is presented. The construction algorithm creates a TCA whose size is in the worst case linear with respect to the size of the indexed tree(s). In the best case, the size of the created TCA is logarithmic. This property of TCA can be used for compression of the indexed tree(s).

A TCA for a tree with n nodes is shown to have at most $n + 1$ states, $2n + 1$ pushdown store symbols and the number of transition rules is $4n$. If a hash map is used for the storage of the transition function of a TCA, the construction of a TCA for a tree with n nodes takes time $\mathcal{O}(2n)$ and requires working space of size at most $2n$.

A linear-time decompression algorithm for TCA is presented. The compression and decompression performance of TCA is verified experimentally and compared to other compression methods. A library that provides compression by TCA is available [52].

An algorithm for subtree matching that uses TCA is introduced. Given a tree t with n nodes and a set of trees T, the algorithm reports all subtrees of tree t that match trees in set T in time $\mathcal{O}(2n)$ if hashing is used.

An algorithm for finding exact repeats of subtrees in a set of trees is presented. The algorithm for finding exact repeats takes linear time with respect to the size of the input when a hash map is used for the storage of transition function δ.

A conversion algorithm from a TCA to a corresponding deterministic FTA that accepts the same trees is given.

1.2.2 A full and linear index of a tree for tree patterns

Whereas the pushdown automaton is a convenient computational model for indexing of a tree for subtrees, it quickly becomes unwieldy when faced with indexing of a tree for tree patterns [3]. This work shows that if used for indexing of a tree for tree patterns, the deterministic pushdown automaton offered by arbology [32] and deterministic finite tree automata used for the same purpose have an exponential worst-case size with respect to the size of the indexed tree.

This motivated the creation of a linear size index of a tree for tree patterns. The index consists of a compact suffix automaton [20], and a *subtree jump table.*

Given a subject tree t with n nodes, the indexing phase is proved to take $\mathcal{O}(n)$ time and require $\mathcal{O}(n)$ space. The number of distinct tree patterns which match the tree is $\mathcal{O}(2^n)$, but the index that is built during the indexing phase requires only $\mathcal{O}(n)$ space.

The searching phase reads an input tree pattern P of size m and locates all its occurrences in tree t. For an input tree pattern P in linear prefix notation $pref(P) = P_1SP_2S\dots SP_k$, $k \geq 1$, the searching is performed in time $\mathcal{O}(m + \sum_{i=1}^{k} |occ(P_i)|))$, where $occ(P_i)$ is the set of all occurrences of string P_i in $pref(t)$.

It is shown that searching for tree patterns using the linear index is analogous to a simulation of a tree pattern PDA.

1.2.3 On space requirements of indexes of a tree for tree patterns based on FTA and on tree pattern PDA

There exist indexes of trees for tree patterns built on top of pushdown automata and on top of finite tree automata whose searching time is strictly linear with respect to the searched pattern. Their space requirements can be exponential, though.

The deterministic tree pattern pushdown automaton [50] is such an index. The searching time is linear if hashing is used for the storage of the transition function. For some trees, however, the deterministic tree pattern PDA has an exponential size.

An index of a tree for tree pattern may also be built as a finite tree automaton. An algorithm that builds such an index is presented. It is proved that there are trees such that any deterministic FTA that indexes them has exponential size, whereas the non-deterministic FTA that indexes them has linear size.

1.3 Structure of the doctoral thesis

This doctoral thesis is organised into 7 chapters as follows:

1. *Introduction*: Describes the motivation behind the efforts of the doctoral thesis together with its goals. It places this work in the context of arbology. It identifies two areas suitable for further research: a compressed indexing of a tree and an indexing of a tree for tree patterns.

2. *Definitions*: Introduces the reader to the necessary theoretical background. It presents the necessary pushdown automata from arbology [3] and the finite tree automaton [18].

3. *Related Results*: Overviews related results from the fields of arbology and stringology, finite tree automata and XML indexing.

4. *Tree Compression Automaton*: This section presents tree compression automaton (TCA) and the algorithms for tree indexing, tree pattern matching, subtree repeats searching and tree compression that use TCA.

5. *A Full and Linear Index of a Tree for Tree Patterns*: This section presents a method that constructs a linear-sized index of a tree that is subsequently used for tree pattern (tree template) searching. The method is compared with related results, including a detailed space and time complexity analysis.

6. *On Space Requirements of an Index of a Tree for Tree Patterns*: This section analyzes the space required for building an index of a tree for tree patterns. Two methods for building the index are analyzed; one uses a finite tree automaton, the other a tree pattern PDA. It is proved that an index in the form of a finite tree automaton or in the form of a tree pattern PDA requires a worst-case exponential size with respect to the size of the indexed tree. Examples of trees for which the indexes have exponential size are shown. On the other hand, examples of trees for which the indexes have linear size are also shown. Several properties of trees that affect the size of the indexes are considered.

7. *Conclusions*: This section summarises the results of the research, suggests possible topics of the further research, and concludes the doctoral thesis.

Chapter 2

Definitions

2.1 Basic Definitions

This section defines the basic terms necessary for the doctoral thesis. The definitions of the basic notions are partially based on [32] and [18]. The following theoretical concepts are introduced: alphabet, language, context–free grammar, pushdown automaton, graph, tree, prefix notation, tree bar notation, finite tree automaton.

2.1.1 Alphabet, language, context–free grammar, pushdown automaton

Notions from the theory of string languages are defined similarly as they are defined in [1].

An *alphabet* is a nonempty finite set of symbols. A *language* over an alphabet $\mathcal{A}$ is a set of strings over $\mathcal{A}$. Expression $\mathcal{A}^*$ stands for the set of all strings over $\mathcal{A}$ including the empty string, denoted by ε. Set $\mathcal{A}^+$ is defined as $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. Similarly, for string $x \in \mathcal{A}^*$, notation x^m, $m \geq 0$, denotes the m-fold concatenation of x with $x^0 = \varepsilon$. Set x^* is defined as $x^* = \{x^m : m \geq 0\}$ and $x^+ = x^* \setminus \{\varepsilon\} = \{x^m : m \geq 1\}$. Given a string x, $|x|$ denotes the length of x.

A *context-free grammar* (CFG) is a 4-tuple $G = (N, \mathcal{A}, P, S)$, where N and $\mathcal{A}$ are finite disjoint sets of *non-terminal* and *terminal symbols*, respectively. P is a finite set of *rules* of the form $A \to \alpha$, where $A \in N$, $\alpha \in (N \cup \mathcal{A})^*$. $S \in N$ is the *start symbol.* Relation $\Rightarrow$ is called *derivation*: if $\alpha A \gamma \Rightarrow \alpha\beta\gamma$, $A \in N$, and α, β, $\gamma \in (N \cup \mathcal{A})^*$, then rule $A \to \beta$ is in P. Symbols $\Rightarrow^+$, and $\Rightarrow^*$ are used for the *transitive* closure, and the *transitive and reflexive* closure of $\Rightarrow$, respectively. The language generated by a G, denoted by $L(G)$, is the set of strings $L(G) = \{w : S \ \Rightarrow^* w,\ w \in \mathcal{A}^*\}$.

An (extended) *non-deterministic pushdown automaton* (non-deterministic PDA) is a seven-tuple $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$, where Q is a finite set of *states*, $\mathcal{A}$ is an *input alphabet*, G is a *pushdown store alphabet*, δ is a transition function from $Q \times (\mathcal{A} \cup \{\varepsilon\}) \times G^*$ into a set of finite subsets of $Q \times G^*$, $q_0 \in Q$ is an initial state, $Z_0 \in G$ is the initial pushdown store symbol, and $F \subseteq Q$ is the set of final (accepting) states. Elements of the transion function will be called *transition rules.* Triple $(q, w, x) \in Q \times \mathcal{A}^* \times G^*$ denotes the configuration of a pushdown automaton. In this doctoral thesisthe top of the pushdown store x is written on its left hand side. The initial configuration of a pushdown automaton is a triple (q_0, w, Z_0) for the input string $w \in \mathcal{A}^*$.

The relation $\vdash_M \subset (Q \times \mathcal{A}^* \times G^*) \times (Q \times \mathcal{A}^* \times G^*)$ is a *transition* of a pushdown automaton M. It holds that $(q, aw, \alpha\beta) \vdash_M (p, w, \gamma\beta)$ if $\delta(q, a, \alpha) \ni (p, \gamma)$. A transition $\vdash_M \subset (Q \times \varnothing \times G^*) \times (Q \times \mathcal{A}^* \times G^*)$ is called an *ε-transition.* The k-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively. A pushdown automaton M is a *deterministic* pushdown automaton (deterministic PDA), if it holds:

1. $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in \mathcal{A} \cup \{\varepsilon\}$, $\gamma \in G^*$.
2. If $\delta(q, a, \alpha) \neq \varnothing$, $\delta(q, a, \beta) \neq \varnothing$ and $\alpha \neq \beta$ then α is not a suffix of β and β is not a suffix of α.
3. If $\delta(q, a, \alpha) \neq \varnothing$, $\delta(q, \varepsilon, \beta) \neq \varnothing$, then α is not a suffix of β and β is not a suffix of α.

A language L accepted by a pushdown automaton M is for the purposes of this doctoral thesis defined by:

1. *Accepting by empty pushdown store:*

$$L_\varepsilon(M) = \{x : (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon) \wedge x \in \mathcal{A}^* \wedge q \in Q\}.$$

When PDA accepts the language by empty pushdown store, the set F of final states is the empty set.

A *compact suffix automaton* M_c for string x is a variant of finite automaton [1] introduced in [20]. The compact suffix automaton accepts any sub-string w of string x in time $\mathcal{O}(|w|)$ [20]. Construction of the compact suffix automaton for string x takes time $\mathcal{O}(|x|)$ [20]. The automaton reports all k occurrences of a sub-string w in string x in time $\mathcal{O}(|w|)$ [20].

2.1.2 Graph, tree, prefix notation, bar notation

Notions on trees are defined similarly as they are defined in [1, 18, 35] and [32].

A *ranked alphabet* is a finite nonempty set of symbols each of which has a unique nonnegative *arity* (or *rank*). Given a ranked alphabet $\mathcal{A}$, the arity of a symbol $a \in \mathcal{A}$ is denoted $Arity(a)$. The set of symbols of arity p is denoted by $\mathcal{A}_p$. Elements of arity $0, 1, 2, \ldots, p$ are respectively called nullary (constants), unary, binary, …, p-ary symbols. We assume that $\mathcal{A}$ contains at least one constant. In the examples we use numbers at the end of the identifiers for a short declaration of symbols with arity. For instance, $a2$ is a short declaration of a binary symbol a. We use $|\mathcal{A}|$ notation for the size of set $\mathcal{A}$.

Based on concepts from graph theory (see [1]), a labelled, ordered tree over an alphabet $\mathcal{A}$ can be defined as follows:

An *ordered directed graph* G is a pair (N, R), where N is a set of nodes and R is a set of linearly ordered lists of edges such that each element of R is of the form

$((f, g_1), (f, g_2), \dots, (f, g_n))$, where $f, g_1, g_2, \dots, g_n \in N$, $n \geq 0$. This element will indicate that, for node f, there are n edges leaving f, the first entering node g_1, the second entering node g_2, and so forth.

A sequence of nodes $(f_0, f_1, \dots, f_n)$, $n \geq 1$, is a *path* of length n from node f_0 to node f_n if there is an edge which leaves node f_{i-1} and enters node f_i for $1 \leq i \leq n$. A *cycle* is a path $(f_0, f_1, \dots, f_n)$, where $f_0 = f_n$. An ordered *dag* (dag stands for Directed Acyclic Graph) is an ordered directed graph that has no cycle. A *labelling* of an ordered graph $G = (A, R)$ is a mapping of A into a set of labels. We use a_f for a short declaration of node f labelled by symbol a.

Given a node f, its *out-degree* is the number of distinct pairs $(f, g) \in R$, where $g \in A$. By analogy, the *in-degree* of node f is the number of distinct pairs $(g, f) \in R$, where $g \in A$.

A *rooted and directed tree* t is an acyclic connected directed graph $t = (N, R)$ with a special node $r \in N$, called the *root*, such that

(1) r has in-degree 0,

(2) all other nodes of t have in-degree 1,

(3) there is just one path from the root r to every $f \in N$, where $f \neq r$.

A node g is a *direct descendant* of node f if a pair $(f, g) \in R$. Nodes with out-degree 0 are called *leaves*.

A *labelled, (rooted, directed) tree* is a tree having the following property:

(4) every node $f \in N$ is labelled by a symbol $a \in \mathcal{A}$, where $\mathcal{A}$ is an alphabet.

A *ranked, (labelled, rooted, directed) tree* is a tree labelled by symbols from a ranked alphabet and out-degree of a node f labelled by symbol $a \in \mathcal{A}$ equals to $Arity(a)$. Nodes labelled by nullary symbols (constants) are called *leaves*.

An *ordered, (ranked, labelled, rooted, directed) tree* is a tree where direct descendants $a_{f1}, a_{f2}, \dots, a_{fn}$ of a node a_f having an $Arity(a_f) = n$ are ordered.

A *subtree* of tree $t = (N, R)$ rooted at node $f, f \in N$, is a tree $t_f = (N_f, R_f), N_f \subseteq N, R_f \subseteq R$, such that

1. node $f, f \in N_f$, is the root of t_f,
2. there exists no tree $t' = (N', R'), f \in N', N' \subseteq N, R' \subseteq R$, rooted at node f such that $|N'| > |N_f]$ or $|R'| > |R_f|$.

If g is not a node of an ordered tree (N, R), but not its root, then there exists an edge $(f, g) \in R$. A *right sibling* of node g is a node h that is the smallest node greater than g that satisfies $(f, h) \in R$.

A tree s with root r_s is a *child subtree* of tree $t = (N, R)$ with root r_t if s is its subtree and $(r_t, r_s) \in R)$.

Two trees t, t' are identical if their roots r, r' are labeled with the same label, the roots have the same number k of child subtrees s_i, s'_i for $1 \leq i \leq k$ and every two child subtrees s_i, s'_i for $1 \leq i \leq k$ are identical.

The *prefix notation* $pref(t)$ of tree t is defined as follows:

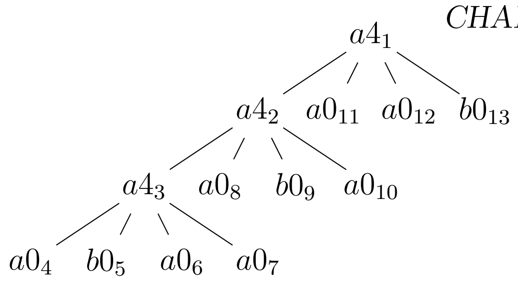


Figure 2.1: Tree t_1 from Example 2.1

1. $pref(a) = a_0$ if a is a leaf,
2. $pref(t) = a_n\ pref(b_1)\ pref(b_2) \ldots pref(b_n)$, where a_n is the root of tree t and $b_1, b_2, \ldots\ b_n$ are the respective child subtrees of a.

The *prefix parenthesis notation pre_par*(t) of tree t is defined as follows:

1. $pref\_par(a) = a$ if a is a leaf,
2. $pref(t) = a(pref\_par(b_1) \ldots pref\_par(b_n))$, where a is the root of tree t and $b_1, b_2, \ldots\ b_n$ are the respective child subtrees of a.

Example 2.1 Consider a ranked alphabet $\mathcal{A} = \{a4,\ a0,\ b0\}$. The number in the symbol stands for the arity of the symbol; thus $Arity(a4) = 4$ and $Arity(b0) = 0$. Consider an ordered, ranked, labelled, rooted, and directed tree $t_1 = (\{a4_1, a4_2, a4_3, a0_4, b0_5, a0_6, a0_7, a0_8, b0_9, a0_{10}, a0_{11},\ a0_{12}, b0_{13}\},\ R_1)$ over an alphabet $\mathcal{A}$, where R_1 is a set of the following ordered pairs:

$$R_1 = \{(a4_1, a4_2), (a4_1, a0_{11}), (a4_1, a0_{12}), (a4_1, b0_{13}), (a4_2, a4_3), (a4_2, a0_8), \\ (a4_2, b0_9), (a4_2, a0_{10}), (a4_3, a0_4), (a4_3, b0_5), (a4_3, a0_6), (a4_3, a0_7)\}.$$

Prefix notation of tree t_1 is $pref(t_1) = a4_1a4_2a4_3a0_4b0_5a0_6a0_7a0_8b0_9a0_{10}a0_{11}\ a0_{12}b0_{13}$. Tree t_1 is illustrated in Figure 2.1.

We will use notation $|t|$ to denote the number of nodes of a tree t.

To define a *tree pattern*, we use a special nullary symbol $S \notin \mathcal{A}$, $Arity(S) = 0$, which serves as a placeholder for any subtree. A tree pattern is defined as a labelled ordered tree over an alphabet $\mathcal{A} \cup \{S\}$. In this doctoral thesis we assume that the tree pattern has at least one node labelled by a symbol from $\mathcal{A}$.

A tree pattern p with $k \geq 0$ occurrences of symbol S *matches* a subject tree t (at node n) if there exist subtrees $t_1, t_2, \ldots, t_k$ (not necessarily the same) of tree t such that tree p', obtained from tree pattern p by substituting subtree t_i for the i-th occurrence of symbol S in p, $i = 1, 2, \ldots, k$, is equal to the subtree t_s of tree t rooted at node n. Tree t_s is the *matched subtree* of tree t.

Let a tree pattern p match a subject tree t at node n and let m be the number of nodes in the matched subtree t_s. Let i be the index of node n in $pref(t) =$

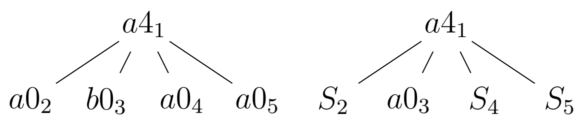


Figure 2.2: Subtree p' (left) and tree pattern p'' (right) from Example 2.2

$a_1a_2\dots a_ia_{i+1}\dots a_{i+m-1}a_{i+m}\dots$. An *occurrence* of tree pattern p in subject tree t is a pair $(i, i+m)$. The pair $(i, i+m)$ is also an *occurrence* of sub-string $pref(t_s)$ in string $pref(t)$.

Example 2.2 Consider tree t_1 from Example 2.1.

Consider subtree p' over alphabet $\mathcal{A}, p' = (\{a4_1, a0_2, b0_3, a0_4, a0_5\}, R_{p'})$, $pref(p') = a4\ a0\ b0\ a0\ a0$ and $R_{p'} = \{((a4_1, a0_2), (a4_1, b0_3)), ((a4_1, a0_4), ((a4_1, a0_5))\}$.

Consider tree pattern p'' over an alphabet $\mathcal{A} \cup \{S\}$, $p'' = (\{a4_1,\ S_2,\ a0_3,\ S_4,\ S_5\}, R_{p''})$. Tree pattern p'' in prefix notation is $pref(p'') = a4\ S\ a0\ S\ S$ and $R_{p''} = \{((a4_1, S_2), (a4_1, a0_3)), ((a4_1, S_4), ((a4_1, S_5))\}$.

Tree patterns p' and p'' are illustrated in Figure 2.2. Tree pattern p' has one occurrence in tree t_1 – it matches t_1 at node $a4_3$. Tree pattern p'' has two occurrences in tree t_1 – it matches t_1 at nodes $a4_1$ and $a4_2$.

The *prefix bar notation* $pref\_bar(t)$ of tree t is defined as follows:

1. $pref\_bar(a) = a \mid$ (a is a leaf),
2. $pref\_bar(t) = a\ pref\_bar(b_1)\ pref\_bar(b_2)\dots pref\_bar(b_n) \mid$, where a is the root of tree t and $b_1, b_2, \dots\ b_n$ are direct descendants of a.

2.1.3 Index of a tree

An *index* is a structure that extracts information from data to improve the query times over it. For an index, the following are the fundamental considerations:

– type of the supported queries,

– *size* of the index relative to data size,

– query time.

Assume a given subject tree t. An *index of tree t for subtrees*, or a full index that accepts all subtrees of tree t, is a structure based on tree t that allows for fast (typically sub-linear or independent of the size of tree t) queries about the presence of a given subtree within tree t. An *index of tree t for tree patterns*, or a full index that accepts all tree patterns that match tree t at any of its nodes, is a structure based on tree t that allows for fast queries about the presence of subtrees of t that are matched by a given tree pattern at their root node.

2.2 Finite tree automaton

Tree compression automaton (TCA), which is presented in the next chapter, is closely related to finite tree automaton (FTA) [18]. The theory of regular tree languages views sets of trees as languages, some of which are regular (or recognizable, the notions are shown to be equivalent in [18]). Regular tree languages are generated by regular tree grammars and they are accepted by finite tree automata. The theory shows that regular tree languages share a number of properties with context-free languages. One of the most important is the fact that the set of derivation trees of any context-free language is a regular tree language. Another important property is that given a regular tree language, the set of yields of its elements is a context-free language. The following text presents these concepts formally. The definitions are based on [18].

The first definition introduces finite tree automaton. For representing trees, the definition uses prefix parenthesis notation. It uses the notion of *variable*, which in this context is a constant that stands for any subtree. For FTA, it is customary to express a tree in notation $a(bcd)$, where a is the root of the tree and b, c, d are the subtrees rooted under a.

Definition 2.3 Let $\mathcal{X}$ be a set of *variables*. Let $\mathcal{A}$ be a ranked alphabet. A (nondeterministic) *finite tree automaton* (NFTA) over $\mathcal{A}$ is a 4-tuple $M = (Q, \mathcal{A}, Q_f, \delta)$ where Q is a set of (unary) states, $Q_f \subseteq Q$ is a set of final states, and δ is a set of transition rules of the following type:

$$f(q_1(x_1) \dots q_n(x_n)) \to q(f(x_1 \dots x_n))),$$

where $n \geq 0$, $f \in \mathcal{A}_n$, $q, q_1, \dots, q_n \in Q$, $x_1, ..., x_n \in \mathcal{X}$.

Example 2.4 Let $\mathcal{A} = \{a2, a1, a0\}$. Consider NFTA $M = (Q, \mathcal{A}, Q_f, \delta)$, where $Q = \{q_{odd}, q_{even}\}$, $Q_f = \{q_{even}\}$, and

$$\begin{array}{rcll}
\delta = \{\ a0 & \to & q_{odd}(a0), & (1)\\
a1(q_{odd}(x)) & \to & q_{odd}(a1(x)), & (2)\\
a1(q_{even}(x)) & \to & q_{even}(a1(x)), & (3)\\
a2(q_{odd}(x)\ q_{odd}(y)) & \to & q_{even}(a2(x\ y)), & (4)\\
a2(q_{even}(x)\ q_{even}(y)) & \to & q_{even}(a2(x\ y)), & (5)\\
a2(q_{odd}(x)\ q_{even}(y)) & \to & q_{odd}(a2(x\ y)), & (6)\\
a2(q_{even}(x)\ q_{odd}(y)) & \to & q_{odd}(a2(x\ y))\ \}. & (7)
\end{array}$$

This automaton accepts all binary trees with an even number of leaves.

A finite tree automaton accepts trees over $\mathcal{A}$. The automaton runs in bottom-up way. There is no initial state; instead, the automaton applies its transition function δ whenever it is applicable to any of its subtrees. For this purpose, a *tree context* and a *move relation* have to be defined.

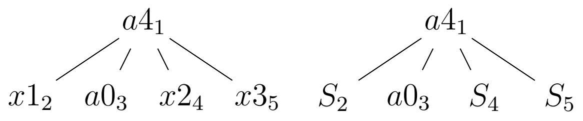


Figure 2.3: Tree context C' from example 2.6 (left) and tree pattern p'' (right) from Example 2.2

Definition 2.5 Let $\mathcal{A}$ be an alphabet. Let $\mathcal{X} = \{x1, \dots, xn\}$ be a set of variables. A *tree context* C over $\mathcal{A} \cup \mathcal{X}$ is a tree over alphabet $\mathcal{A} \cup \mathcal{X}$. The expression $C[t_1, \dots, t_n]$ denotes a tree t obtained from C, in which each variable xi for all $0 < i \leq n$ has been replaced by tree t_i. A tree context in which each variable occurs at most once is called *linear*. Tree contexts that are not linear are called *non-linear*.

In this text, only linear tree contexts are considered.

Example 2.6 Let $\mathcal{A} = \{a4, a0\}$. Let $\mathcal{X} = \{x1, x2, x3\}$. Let C' be a tree context over $\mathcal{A} \cup \mathcal{X}$, $pref(C') = a4_1x1_2a0_3x2_4x3_5$. Context C' is illustrated in Figure 2.3. Compare the tree context with tree pattern p'' from Example 2.2. The semantics of both patterns is analogous. Whereas every xi represents a unique subtree, the S placeholder symbol can stand for any subtree at each of its occurrences. Since the symbols S do not allow non-linear pattern matching, tree contexts where any variable xi occurs more than once are not considered.

Definition 2.7 Let $M = (Q, \mathcal{A}, Q_f, \delta)$ be a finite tree automaton over $\mathcal{A}$. Let t, t' be trees over $\mathcal{A} \cup Q$. Let $T(\mathcal{A} \cup Q)$ be the set of all trees over $\mathcal{A} \cup Q$. The relation $\underset{M}{\rightarrow} \subset T(\mathcal{A} \cup Q) \times T(\mathcal{A} \cup Q)$ is called a *move relation* and an application of this move relation is called a *reduction*. It holds that

$$t \underset{M}{\rightarrow} t' \iff \begin{cases} \exists \text{ context } C \text{ over } \mathcal{A} \cup Q, \exists \text{ trees } t_1, \dots, t_n, \\ f(q_1(x_1) \dots q_n(x_n)) \rightarrow q(f(x_1 \dots x_n)) \in \delta, \\ t = C[f(q_1(t_1), \dots, q_n(t_n))], \\ t' = C[q(f(t_1 \dots t_n))]. \end{cases}$$

$\overset{*}{\underset{M}{\rightarrow}}$ denotes the reflexive and transitive closure of $\underset{M}{\rightarrow}$. Automaton M *accepts* tree t if $t \overset{*}{\underset{M}{\rightarrow}} q(t)$ and $q \in Q_f$.

Example 2.8 Consider NFTA M from Example 2.4. Consider tree t_2, $pref(t_2) = a2_1a1_2a0_3a0_4$. Tree t_2 is illustrated in Figure 2.4. It holds that $t_2 \overset{*}{\underset{M}{\rightarrow}} q_{even}(t_2)$. The series of reductions is shown in Figure 2.5. Automaton M ends in state q_{even}, which if final. The tree is accepted.

Definition 2.9 A *deterministic* finite tree automaton (DFTA) is a non-deterministic finite tree automaton in which no two transition rules have the same left-hand side.

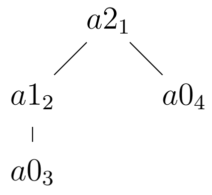


Figure 2.4: Tree t_2 from Example 2.8

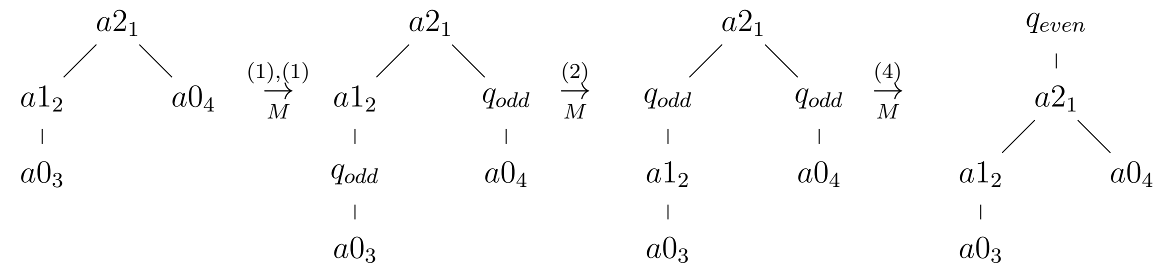


Figure 2.5: Reductions that NFTA M from Example 2.4 performs on tree t_2 from Example 2.8

Example 2.10 The finite tree automaton from Example 2.4 is deterministic.

A set of trees is a *tree language*. Two FTAs are equivalent if they accept the same tree languages.

For every NFTA there exists an *equivalent* DFTA. The equivalent DFTA may be constructed by a determinisation algorithm [18].

The NFTAs can be extended with *ε-transitions*, similar to ε-transitions of string finite automata. Such extended NFTAs are of the same power as plain NFTAs and a conversion algorithm exists [18].

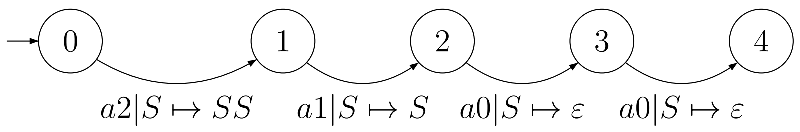
For every set L of trees recognized by some NFTA there exists a *minimum* DFTA that accepts it. This minimum DFTA has the least number of states of all DFTAs that accept set L.

A *complete* DFTA defines its transition function for all possible inputs [18].

2.3 Subtree PDA, Tree Pattern PDA

The subtree pushdown automaton and the tree pattern pushdown automaton are now presented, based on definitions from [50].

Definition 2.11 Let t and $pref(t)$ be a tree and its prefix notation, respectively. A *subtree pushdown automaton* for $pref(t)$ is a pushdown automaton $M_{nps}(t)$ that accepts all subtrees of t in prefix notation [50].



$$\begin{tikzcd}
\arrow[r] & 0 \arrow[r, bend right, "a2|S \mapsto SS"'] & 1 \arrow[r, bend right, "a1|S \mapsto S"'] & 2 \arrow[r, bend right, "a0|S \mapsto \varepsilon"'] & 3 \arrow[r, bend right, "a0|S \mapsto \varepsilon"'] & 4
\end{tikzcd}$$

Figure 2.6: Transition diagram of PDA $M_p(t_2)$ from Example 2.13 constructed for tree t_2 from Example 2.8

The construction of subtree pushdown automaton is done in two steps. In the first step, a pushdown automaton that accepts $pref(t)$ is constructed by algorithm 2.12. In the second step, the automaton is extended into a subtree pushdown automaton by algorithm 2.14.

Algorithm 2.12: Construction of a PDA accepting $pref(t)$ by empty pushdown store [50]

```
  Name: Subtree-PDA-preparation
  Input: A tree $t$ over ranked alphabet $\mathcal{A}$ in prefix notation $pref(t) = a_1 a_2 \dots a_n$,
         $n \geq 1$
  Output: Pushdown automaton $M_p(t) = (\{0, 1, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$
1 begin
2     foreach state $i$, where $1 \leq i \leq n$ do
3         insert into $\delta$ the following transition rule: $\delta(i-1, a_i, S) = (i, S^{Arity(a_i)})$,
            where $S^0 = \varepsilon$
4     end
5 end
```

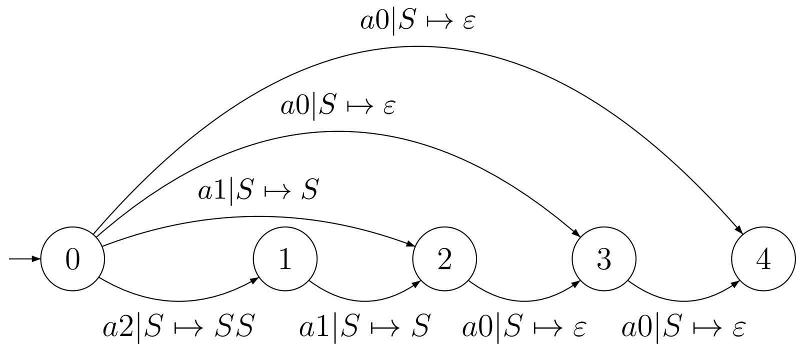
Example 2.13 Consider tree t_2 from Example 2.8, $pref(t_2) = a2_1 a1_2 a0_3 a0_4$. The automaton $M_p(t_2) = (\{0, 1, 2, 3, 4\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$ accepting tree t_2 has its transition function δ illustrated in Figure 2.6.

Example 2.15 Consider tree t_2 from Example 2.8, $pref(t_2) = a2_1 a1_2 a0_3 a0_4$. The automaton $M_{nps}(t_2) = (\{0, 1, 2, 3, 4\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$ constructed for tree t_2 has its transition function δ illustrated in Figure 2.7.

Definition 2.16 Let t and $pref(t)$ be a tree and its prefix notation, respectively. A *tree pattern pushdown automaton* for $pref(t)$ is a pushdown automaton that accepts all tree patterns in prefix notation which match tree t at some of its nodes [50].

Algorithm 2.14: Construction of a non-deterministic subtree PDA [50]

```
  Name: Subtree-PDA-construction
  Input: A tree $t$ over ranked alphabet $\mathcal{A}$ in prefix notation $pref(t) = a_1 a_2 \dots a_n$,
         $n \geq 1$
  Output: Non-deterministic PDA $M_{nps}(t) = (\{0, 1, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$
1 begin
2     create PDA $M_p(t)$ by Algorithm 2.12 (Subtree-PDA-preparation) and set
        $M_{nps}(t) = M_p(t)$;
3     foreach state $i$, where $2 \leq i \leq n$ do
4         insert into $\delta$ the following transition rule: $\delta(0, a_i, S) \ni (i, S^{Arity(a_i)})$, where
            $S^0 = \varepsilon$
5     end
6 end
```



$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
0
1
2
3
4
$a2|S \mapsto SS$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$

Figure 2.7: Transition diagram of PDA $M_{nps}(t_2)$ from Example 2.15 constructed for tree t_2 from Example 2.8

The construction of tree pattern pushdown automaton is in two steps, similarly to construction of subtree pushdown automaton. In the first step a special automaton is constructed and has to be defined.

Definition 2.17 Let t, r and $pref(t)$ be a tree, its root and its prefix notation, respectively. A *treetop pushdown automaton* for $pref(t)$ accepts all tree patterns in prefix notation which match the tree t at root r [50].

Definition 2.18 Let t and $pref(t)$ be a tree and its prefix notation, respectively. The set $srms(t)$ of *subtree rightmost states* is defined as $srms(t) = \{i : pref(t) = a_1 \ldots a_n, 1 \le i \le n, Arity(a_i) = 0\}$ [50].

Algorithm 2.19: Construction of a treetop PDA for a tree t in prefix notation $pref(t)$ [50]

```
  Name: Treetop-PDA-construction
  Input: A tree $t$ over ranked alphabet $\mathcal{A}$ in prefix notation $pref(t) = a_1 a_2 \ldots a_n$,
         $n \ge 1$
  Output: Treetop PDA $M_{pt}(t) = (\{0, 1, \ldots, n\}, \mathcal{A} \cup \{S\}, \{S\}, \delta, 0, S, \emptyset)$
1 begin
2     create PDA $M_p(t)$ by Algorithm 2.12 (Subtree-PDA-preparation) and set
        $M_{pt}(t) = M_p(t)$;
3     create a set $srms = \{i : 1 \le i \le n, \delta(i-1, a, S) = (i, \varepsilon), a \in \mathcal{A}_0\}$;
4     foreach state $i$, where $i = n-1, n-2, \ldots 1$, $\delta(i, a, S) = (i+1, S^p), a \in \mathcal{A}_p$ do
5         insert into $\delta$ the following transition rule: $\delta(i, S, S) \ni (l, \varepsilon)$, where $l$ is the
            $p$-th smallest integer such that $l \in srms$ and $l > i$;
6         remove all $j$, where $j \in srms$ and $i < j < l$, from $srms$;
7     end
8 end
```

Example 2.20 Consider tree t_2 from Example 2.8. Treetop PDA $M_{pt}(t_2)$ constructed by Algorithm 2.19 (Treetop-PDA-construction) is illustrated in Figure 2.8.

Example 2.22 Consider tree t_2 from Example 2.8. Non-deterministic tree pattern PDA $M_{npt}(t_2)$ constructed by Algorithm 2.21 (Tree-pattern-PDA-construction) is illustrated in Figure 2.9.

The tree pattern PDA M_{npt} may be determinised to a deterministic tree pattern PDA M_{dpt} using a standard determinisation algorithm for input-driven pushdown automata from [50], presented in Algorithm 2.23.

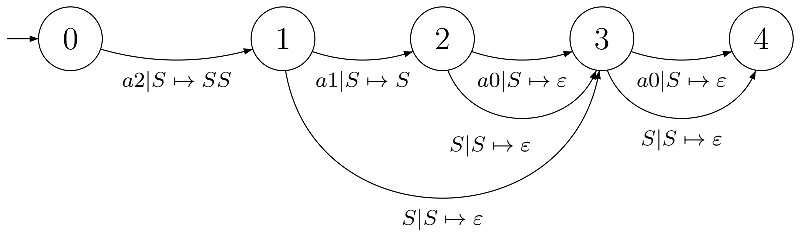

0
1
2
3
4
$a2|S \mapsto SS$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$

Figure 2.8: Transition diagram of treetop PDA $M_{pt}(t_2)$ from Example 2.20 constructed for tree t_2 from Example 2.8

Algorithm 2.21: Construction of a non-deterministic tree pattern PDA for a tree t in prefix notation $pref(t)$ [50]

```
  Name: Tree-pattern-PDA-construction
  Input: A tree $t$ over ranked alphabet $\mathcal{A}$ in prefix notation $pref(t) = a_1a_2 \ldots a_n$,
         $n \geq 1$
  Output: Non-deterministic tree pattern PDA
          $M_{npt}(t) = (\{0, 1, \ldots, n\}, \mathcal{A} \cup \{S\}, \{S\}, \delta, 0, S, \emptyset)$
1 begin
2     create PDA $M_{pt}(t)$ by Algorithm 2.19 and set $M_{npt}(t) = M_{pt}(t)$;
3     foreach state $i$, where $2 \leq i \leq n$ do
4         insert into $\delta$ the following transition rule: $\delta(0, a_i, S) \ni (i, S^{Arity(a_i)})$, where
          $S^0 = \varepsilon$;
5     end
6 end
```

Algorithm 2.23: Construction of an equivalent deterministic PDA for an input-driven non-deterministic PDA [50]. The sets that form states in Q' shall be called *d-subsets*.

```
  Name: Input-driven-PDA-determinisation
  Input: Input-driven non-deterministic PDA $M_n = (\{0, 1, \ldots n\}, \mathcal{A}, G, \delta, 0, Z_0, \emptyset)$
  Output: An equivalent deterministic PDA $M_d = (Q', \mathcal{A}, G, \delta', q_I, Z_0, \emptyset)$
1 begin
2     set $Q' = \{[0]\}$, $q_I = [0]$, $q_I$ is an unmarked state;
3     foreach unmarked state $q'$ from $Q'$ do
4         insert into $\delta$ the following transition rule: $\delta(q', a, \alpha) \ni (q'', \beta)$, where
          $q'' = \{q : \delta(p, a, \alpha) \ni (q, \beta)$ for all $p \in q'\}$;
5         if $q''$ is not in $Q'$, add $q''$ to $Q'$ as unmarked state;
6     end
7 end
```

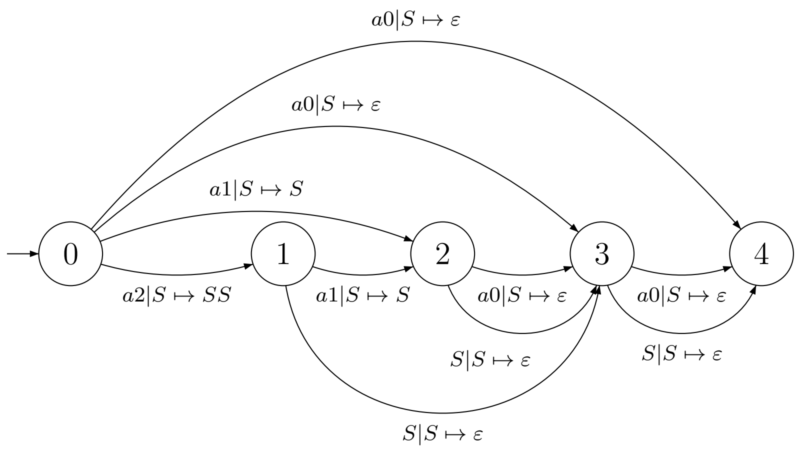

$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
0
1
2
3
4
$a2|S \mapsto SS$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$

Figure 2.9: Transition diagram of tree pattern PDA $M_{npt}(t_2)$ from Example 2.22 constructed for tree t_2 from Example 2.8

Chapter 3

Related Work

3.1 Introduction

Indexing a data subject preprocesses the subject and allows to locate occurrences of input patterns in the subject repeatedly and quickly, in time typically not depending on the size of the subject. The subject is typically much larger than the input patterns.

The similar problem of matching a pattern in a data subject preprocesses the pattern and allows to locate its occurrences in input data subjects repeatedly and quickly, in time typically not depending on the size of the pattern. The pattern is typically much smaller than the subject.

Since trees can be represented as strings, it is appropriate to explore methods that are used for string indexing.

3.2 String indexing

The theory of text indexing, explored extensively in stringology [20, 22, 23], is very well-researched and uses many sophisticated data structures: suffix tree and suffix array are widely used structures for text indexing, providing efficient solutions for a wide range of applications [20]. The Directed Acyclic Word Graph [9], also known as suffix (or factor) automaton [19], is another efficient indexing structure. The minimised versions of suffix trees and suffix automata are represented by compact versions of suffix (factor) automata [8, 20].

The structures presented above share a common property: they index the set of tree's suffixes.

Definition 3.1 Let $w = a_1 a_2 \dots a_n$ be a string over alphabet $\mathcal{A}$. The set of suffixes of w, denoted by $\mathrm{suff}(w)$, is defined as $\mathrm{suff}(w) = \{w_j = a_j a_{j+1} \dots a_n | \text{ for all } 1 \leq j \leq n\}$.

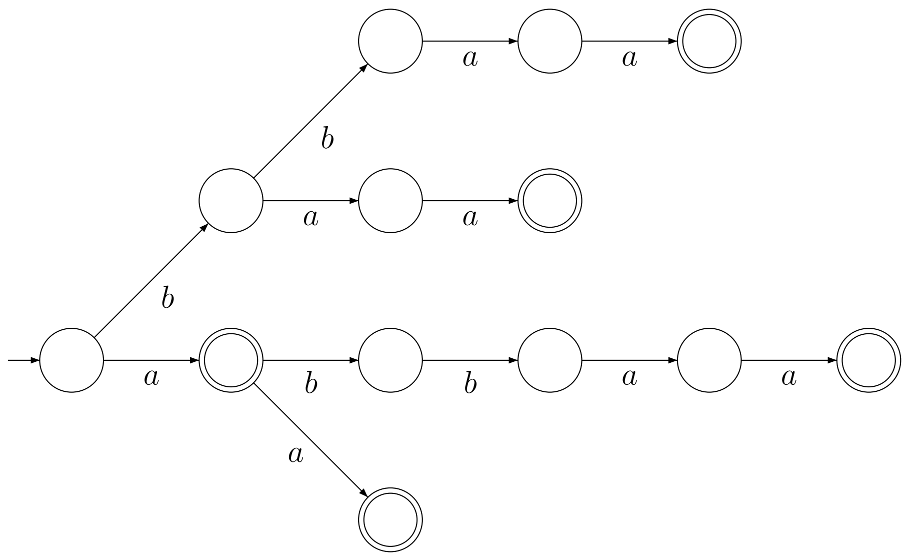


Figure 3.1: Suffix trie of string *abbaa*

3.2.1 Suffix trie

One of the simplest indexes of tree's suffixes is a suffix trie [34]. One of possible definitions of a suffix trie is given below.

Definition 3.2 A *suffix trie* of a string w is a finite automaton that accepts suff(w). It has no unreachable states and no loops. All its non-root states have in-degree 1 and all its leaf states are final states.

The graph defined by the trie's states and transitions is a rooted directed tree.

Example 3.3 Consider string w over alphabet $\{a, b\}$, $w = abbaa$. A suffix trie of string w is presented in Fig. 3.1.

The suffix trie is space-inefficient. It indexes a string of length n into a graph of at most $n * (n + 1)/2 + 1$ nodes.

3.2.2 Suffix tree

A *suffix tree* of a string w [64] is a compacted version of suffix trie of a string w. It collapses the trie by deleting from it all states whose out-degree is 1 and that are not final states.

To describe the conversion algorithm more formally, let q be a state to remove. It has one incoming edge (q_s, q) with a string label x and one outgoing edge (q, q_t) with a string label y. The state is removed along with edges (q_s, q) and (q, q_t) and is replaced by an edge (q_s, q_t) with a label xy. This procedure is repeated until there is no state q left to remove.

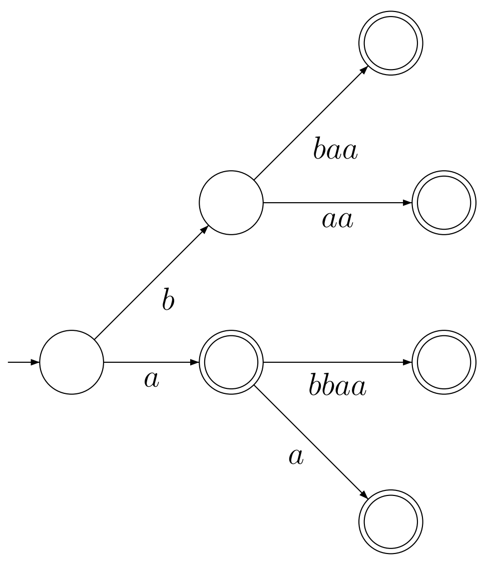


Figure 3.2: Suffix tree of string *abbaa*

The conversion described above is trivial and it creates a suffix tree of string w in time $\mathcal{O}(n^2)$. An on-line algorithm for construction of suffix trees that takes linear time has been presented in [60].

For a string of length n, suffix tree has at between $n+1$ and $2n$ states and edges [20]. It also stores the input string w and each of its edges is represented as a pair (i, d), where i is an index into w and d is a length of a substring from w at index i.

Example 3.4 A suffix tree of string *abbaa* is presented in Fig. 3.2.

3.2.3 Suffix automaton, compact suffix automaton

Another way to decrease the size of a suffix trie is to minimize it. The resulting automaton is called *suffix automaton* or also *Directed Acyclic Word Graph (DAWG)* [24].

Suffix automaton that indexes strings w of length n has at most $2n-1$ nodes and $3n-4$ edges [20].

Example 3.5 A suffix automaton that accepts suff(*abbaa*) is presented in Fig. 3.3.

There is still some redundancy present in the suffix automaton. A *compact suffix automaton* is a modified finite automaton that removes this redundancy. This automaton labels its edges not by symbols, but by strings.

The conversion from suffix automaton to compact suffix automaton is straightforward. All non-final states and non-initial states whose in-degree is 1 and whose out-degree is 1 are collapsed. Formally speaking, consider a suffix automaton M that is to be compacted. Extend its transition function to use strings and not just symbols. Take any non-final and

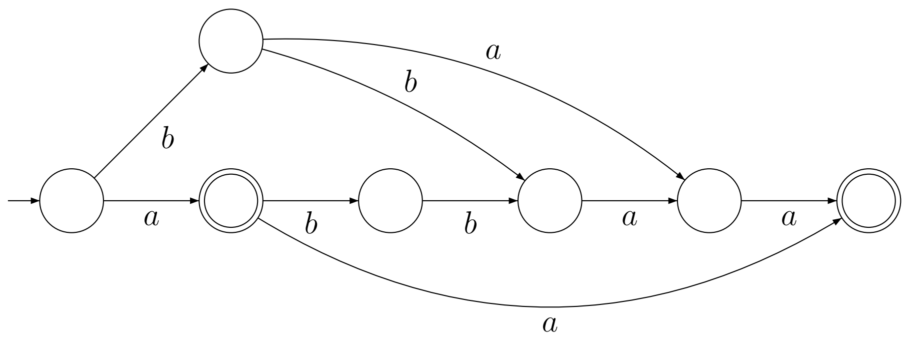


Figure 3.3: Suffix automaton that indexes string *abbaa*

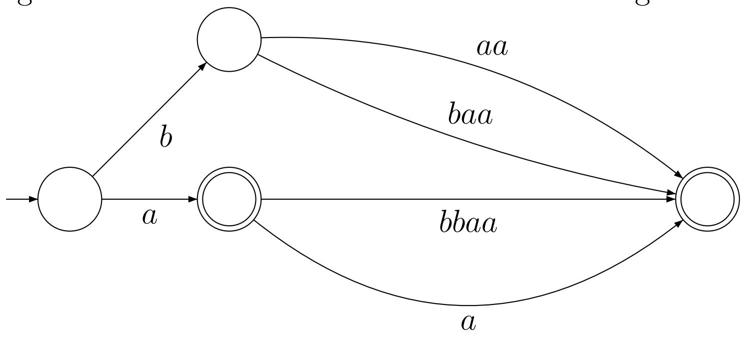


Figure 3.4: Compact suffix automaton that indexes string *abbaa*

non-initial state q of M that has an incoming transition rule (q_s, q) labeled with x and only one outgoing transition rule (q, q_t) labeled with y. Remove the incoming transition rule (q_s, q) and add a new transition rule to the automaton's transition function: $(q_s, xy) \to q_t$. If state q has no incoming transition rules left, remove it. This process is repeated until there are no more states to remove.

The conversion described above is naive and is in fact not necessary for a construction of a compact suffix automaton. A linear-time construction algorithm was presented in [25].

The compact suffix automaton can also be created from suffix tree by minimisation of the suffix tree [20].

Another space-efficient variant to a suffix automaton is a *suffix array* [47], a sorted array of string's suffixes along with possible additional information for faster lookup times.

Example 3.6 A compact suffix automaton that accepts suff($abbaa$) is presented in Fig. 3.4.

3.2.4 String indexing conclusion

Another text indexing structure, called position heap, was proposed recently [27]. It allows building a string's index in linear time and provides searches that depend linearly on the length of the input pattern.

Generally, the number of substrings in a text is quadratic to the size of the text, but the size of the text index structure for substrings is typically linear to the size of the text. This property is desirable for tree indexes. And possible, too: the number of subtrees of a tree is linear to the size of the tree. The size of the index structure is often linear to the size of the tree, but can be smaller in some cases as is shown in [11] and in this doctoral thesis.

3.3 Tree indexing

Trees can be represented as strings by using a linear notation. Subtree searching and tree pattern searching are thus often declared to be analogous to problems of string pattern searching [5, 37]. Still, there are some structural properties of such strings that are unique to linear notations of trees. For example, a prefix linear notation of a tree has an arity checksum of 0 [50]. The similarity between strings and trees means that algorithms used for string indexing can be adapted for indexing trees, as is done by arbology [50].

Trees can be described by means of tree languages and tree automata that accept them. A finite tree automaton [18] is an example of a computational model that relies on tree languages. A finite tree automaton can be used for indexing of a tree for subtrees.

The view of trees as languages suggests another formalism, tree grammars [18]. A grammar-based representation of trees has been found suitable for compression, as shown in [11].

XML is a language used for storing data in a tree-like structure. Since the XML files contain not only a tree but also element data, the methods of XML indexing need to meet a wider set of requirements than do methods used for simple tree indexing. A brief overview of XML indexing methods is provided below.

3.3.1 Arbology and tree indexing

Arbology [50] builds on pushdown automata for processing linearised trees. It has numerous results. Pushdown automaton as a model of computation is suitable for indexing trees [32], finding subtree repeats [31] or searching linear and non-linear patterns in trees [58].

In [50], a pushdown automaton for indexing trees for subtrees was presented under name *subtree pushdown automaton.* This automaton was used as an index for labeled, ordered, ranked trees. The subtree pushdown automaton has both non-deterministic and deterministic variants and accepts a tree and all its subtrees in prefix notation. The construction of the automaton takes $\mathcal{O}(n)$ time in respect to the size of the input tree. An example of a (non-deterministic) subtree pushdown automaton is shown in Fig. 2.7.

The subtree pushdown automaton is directly based on suffix automaton [20]. It is thus not surprising that its determinisation algorithm runs in linear time and the total size of the deterministic automaton is not more than $2n + 1$ states and $3n$ transition rules, given n as the number of nodes of the indexed tree [50].

Finding subtree repeats using subtree pushdown automaton was shown to be possible in $\mathcal{O}(n)$ time and space in [31] and also in [14]. In the latter article the authors did not have to rely on hashing to achieve linear indexing time, the drawback being a need for multiple passes through input. All of the above works rely on ranked trees for input.

3.3.2 Tree indexing for tree patterns

String-based approaches

A tree pattern p is a tree whose leaves can be labelled by a special symbol S, which serves as a placeholder for any subtree. As any other tree, tree pattern p can be converted to string $lin(p)$ using a prefix or postfix linear notation. Consider a subject tree t that contains occurrences of pattern p (symbols S are matched by any subtree). Given a linearized version of a subject tree $lin(t)$, the occurrences of $lin(p)$ correspond to substrings of $lin(t)$. Thus tree pattern searching is analogous to the problem of searching strings (patterns) with gaps.

The problem of searching patterns with gaps and wildcards has been repeatedly explored [28, 2, 21, 33]. The methods differ in the kinds of considered gaps, in the achieved complexity and in the fact whether the subject or the pattern is preprocessed.

In [6], an index is constructed for searching patterns with wildcards. A wildcard matches any single symbol from the alphabet. In [45], an index is constructed for searching patterns with variable length gaps. Unfortunately, the searching time depends on the gap size, which is not efficiently usable for searching tree patterns, where gaps can be of any size.

In [7], a matching algorithm for variable length gap matching in strings was proposed. Even though similar in the definition of the problem, this solution is incompatible with the tree pattern matching problem because of a different interpretation of gaps. Nevertheless, the matching algorithm inspired a method for tree pattern indexing presented in this doctoral thesis.

Tree-based approaches

In [44], a similarly stated problem of tree pattern matching using a non-deterministic pushdown automaton is examined. The solution suffers from an exponential blow-up of number of states of the deterministic pushdown automaton. This problem is partially resolved by using a simulation of the non-deterministic pushdown automaton.

An overview of methods for tree pattern matching and searching is provided in [17]. The work presents an overview of solutions for the problem of tree pattern matching and for the problem of tree acceptance. It defines tree acceptance as the problem of deciding whether a tree belongs to a tree language described by a given tree grammar. The work assembles known solutions into two taxonomies, one for tree pattern matching, the other for tree acceptance.

Tree indexing for tree patterns can be seen as a variant of tree acceptance - the subject tree defines a tree grammar G, which generates all trees and tree patterns that match the subject tree and any of its subtrees. This doctoral thesis focuses on deterministic algorithms for tree indexing based on pushdown automata and tree automata, in a leaf-to-root direction. These are most closely matched by deterministic tree acceptance frontier-to-root algorithms from section 5.6.6.4 of [17]. However, the algorithms from the taxonomy presented there are tailored for general tree grammars, whereas the structure of tree grammar G is very specific. The general deterministic algorithms have a worst-case exponential time and space requirements [17]. This doctoral thesis investigates whether these requirements can be improved upon in case of tree pattern indexing.

In [15] and [30] a non-deterministic pushdown automaton is constructed for tree pattern matching. The solution has a drawback that the equivalent deterministic automaton can have an exponential size.

A finite tree automaton [18] representing a full index of a tree for tree patterns can be constructed, but as this doctoral thesis shows, its size is again exponential with respect to the size of the subject tree.

A subtree pushdown automaton from the arbology research can be extended to accept not only subtrees of a given tree, but also tree patterns that are matched by these subtrees. Such automaton is called *tree pattern pushdown automaton* [50]. This automaton also has non-deterministic and deterministic variants. The deterministic variant can be minimised into a *minimal* tree pattern pushdown automaton, which has the least possible number of states. It is known that the number of distinct tree patterns for a given tree can be exponential, at most $2^{n-1} + n$ [40]. The non-deterministic variant has linear size, whereas the deterministic variant can have size exponential with respect to the non-deterministic counterpart, as is shown in this doctoral thesis. That also means that the determinisation algorithm runs in at worst exponential time. An example of a tree pattern pushdown automaton is shown in Fig. 2.9.

In this thesis, a new method for the indexing of a tree for tree patterns is presented. Given a subject tree t, the tree is preprocessed and an index of tree t, which consists of a standard text compact suffix automaton [20] and a structure called subtree jump table, is constructed. Any text index can possibly be used instead of the compact suffix automaton, which was chosen here because of its convenient space and time complexity.

3.4 Tree compression

Since trees can be represented as strings, known text compression methods can be used for the compression of trees (e.g. [65]). However, when a constraint is placed that the compressed tree structure should act as an index, there are fewer methods available.

In [29], a method for locating subtree repeats was presented. The author correctly observed that the table of tree repeats of a tree can be used for the compression of the processed tree. This fact is used for tree compression by tree compression automaton introduced in Chapter 4.

In [11], grammar-based representation of trees has been found suitable for compression and an algorithm for compression of trees called BPLEX was proposed. The trees can be converted to context-free grammars (also known as straight-line grammars). A straight-line grammar produces a single sentence from its starting symbol. Thus it can represent a tree in its linear form. Moreover, context-free grammars can be extended into *context-free tree grammars* [35].

Context free tree grammars include an additional set Y of parameters beside the set of ranked nonterminal symbols N, the set of terminal symbols T, the set of production rules and the starting symbol. A production rule of a context-free tree grammar has the following form: $A(y_1, \dots, y_k) \to t$, where A is a nonterminal symbol of arity k, $y_1, \dots, y_k$ are parameters from Y and t is a tree over $N \cup T \cup Y$. During rewriting, a parameter may be replaced by a terminal symbol, a non-terminal symbol, or a tree over terminal and non-terminal symbols. If all non-terminal symbols have rank 0, i.e. no parameters appear in production rules, the grammar is a *regular tree grammar*.

The BPLEX algorithm takes as input a straight-line grammar (or an equivalent straight-line regular tree grammar) that generates a tree. It then produces a straight-line context-free tree grammar that generates the same tree using fewer production rules.

A comparison of compression performance between the tree compression automaton from this doctoral thesis and the BPLEX algorithm is provided in section 4.10.

3.5 XML Indexing Methods

XML documents store data in a hierarchical, rooted tree structure. The trees are unordered, which is a difference from the trees considered by algorithms presented in this doctoral thesis. Parsing of the documents can be done in many ways, out of which two have become standard.

An on-line variant, Simple API for XML (or SAX for short) [57] uses an event-based model that allows to process the tree without needing to contain the whole tree in the memory at once. This approach is useful for simpler tasks on XML documents or when the document is too big for the available memory. However, as the tree is never fully in memory, a full index is not built.

A more widespread variant of parsing builds an XML DOM object from the XML Document. The DOM object represents the whole XML tree and is available for queries. The standard language for queries over XML DOM objects [26] is XPath [68] and its superset, XQuery [70], but other query languages are also available (XPointer [69], XLink [67]). XPath is a query language for accessing data in XML documents. It allows searching for specific structures in the trees, similarly to tree pattern matching, but has a greater expressive power. Roughly speaking, it supports queries that can match any subset of XML tree's nodes based on the tree structure and on data stored inside the tree's nodes. XQuery is a superset of XPath; it supports SQL-like queries with FLWOR syntax (FOR-LET-WHERE-ORDERBY-RETURN).

Answering an XPath query often involves traversing the whole XML tree, which is inefficient if done on an unindexed tree. There are many different approaches for building an XML index. In [12], the indexes were classified based on the types of supported queries and the indexing strategy.

The article identified three types of XML queries:

1. Tree structure queries. These queries require a tree index for an efficient execution. The queries define either a straight path through the tree or a branching path through the tree. Tree indexing discussed in this doctoral thesis concentrates on a subset of branching paths (for subtree matching and tree template matching).

2. Queries that start from the root node (total matching) and queries that don't (partial matching). Tree indexing for subtree and tree pattern matching is typically used for partial matching.

3. Content-based queries, which check element content, not the graph structure.

The article also identified two main query processing strategies based on node location schemes they use:

1. Position-based schemes that assign a number to the document's nodes based on their position in the document. These schemes require rebuilding of the index when a new node is inserted.

2. Path-based schemes that add a label to each node. The label captures the path to the node from the root node of the document.

The proposed classification of indexes is the following [12]:

1. Summary indexes. These indexes provide efficient support for non-branching path queries by creating an index as a tree which is created by recursively collapsing the tree's nodes that differ only by order in the document. These indexes require additional processing when presented with a branching query. Typically, the branching queries are de-composed into non-branching queries, whose results are then merged together. Examples include DataGuide [36], Forward and Backward Index [63], Template Index [51], MTree [53], and Adaptive Path Index for XML Data [16]. The last example, APEX Index, is a partial summary index, which means that it indexes only for the most often issued queries.

2. Structural join indexes. These indexes provide quick access to sets of elements or nodes that satisfy elementary queries (usually through B-trees). A query is answered by first being decomposed into a tree of elementary queries, which are resolved using the index and then joined together using a structural join algorithm. Examples include XML Indexing and Storage System [46], XML Region Tree [41] and Lazy XML Join [13].

3. Sequence-based indexes represent the trees in a linearized form, as sequences. Subsequence matching is used for answering queries. The subsequence matching can generate false hits, which have to be pruned by some verification process. Examples include Virtual Suffix Tree [62] and Prüfer sequences [55].

A different classification appears in [49]. Among types of queries it considers are so called "twigs" [10]. The twigs are a generalized form of tree template queries answered by tree template pushdown automaton [30]. A twig describes a subgraph of a tree, which can be composed of multiple (disconnected) components. Compare this with a tree template [30], which always defines a connected subgraph of a tree. The proposed classification separates indexes that support twig queries into sequence-based indexes (Virtual Suffix Tree [62]) and summary-based indexes (Forward and Backward Index [63]). It also adds a new class of indexes, called Content Indexes, which separate storage of data and tree structure (Tindex [48]).

Chapter 4

Tree Compression Automaton

4.1 Definition of tree compression automaton

This chapter presents *tree compression automaton (TCA)*, as described in [A.1]. A *general tree compression automaton (GTCA)* for a set of trees T is a pushdown automaton that accepts all trees in set T and all of their subtrees. Tree compression automaton is defined as a deterministic version of GTCA constructed using the provided construction algorithm. TCA for a set of trees T is proved to be deterministic and to be a GTCA. In the following sections TCA is shown to be suitable for tree indexing, tree matching, tree compression and locating subtree repeats.

Example 4.1 Figure 4.1 shows a tree t_1 and its prefix bar notation.

Definition 4.2 Let T be a set of trees. Let I be a set of symbols. Let μ be an injective mapping from the set of all subtrees of all trees from set T into set I such that two subtrees are assigned the same element from set I if and only if they are identical. The triplet (T, I, μ) is called *subtree identification mapping for set T*.

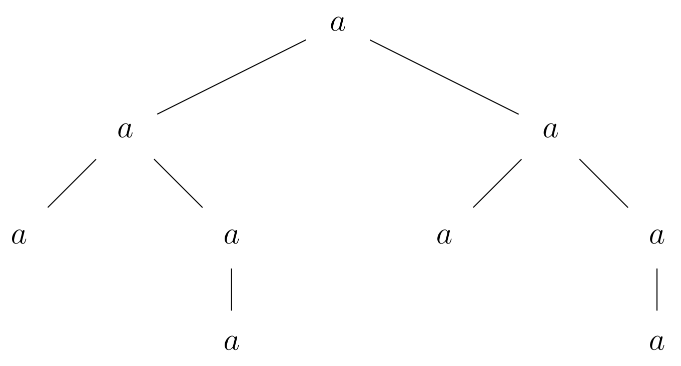
Example 4.3 An example of a subtree identification mapping is shown in Figure 4.2.

Definition 4.4 Let T be a set of trees. Let (T, I, μ) be a subtree identification mapping for set T. Let t be any ordered subtree of any tree in set T. Let it have k child subtrees $child\_subtree_i$ for i from 1 to k in this order. Let r be the root of subtree t and let L be an ordered list $(\mu(child\_subtree_1), \mu(child\_subtree_2), \ldots)$. Given (T, I, μ), the pair (r, L) is called a *tree stub* of tree t.

Example 4.5 Tree stubs that are derived from the subtree identification mapping $(\{t_1\}, \{1, 2, 3, 4\}, \mu_1)$ from Example 4.3:

– tree stub of subtree t_{11}, $t_{11} = a|$, $\mu_1(t_{11}) = 1$: $(a, ())$,

– tree stub of subtree t_{12}, $t_{12} = aa||$, $\mu_1(t_{12}) = 2$: $(a, (1))$,



$$pref\_bar(t_1) = aaa|aa|||aa|aa||||$$

Figure 4.1: Tree t_1 and its prefix bar notation

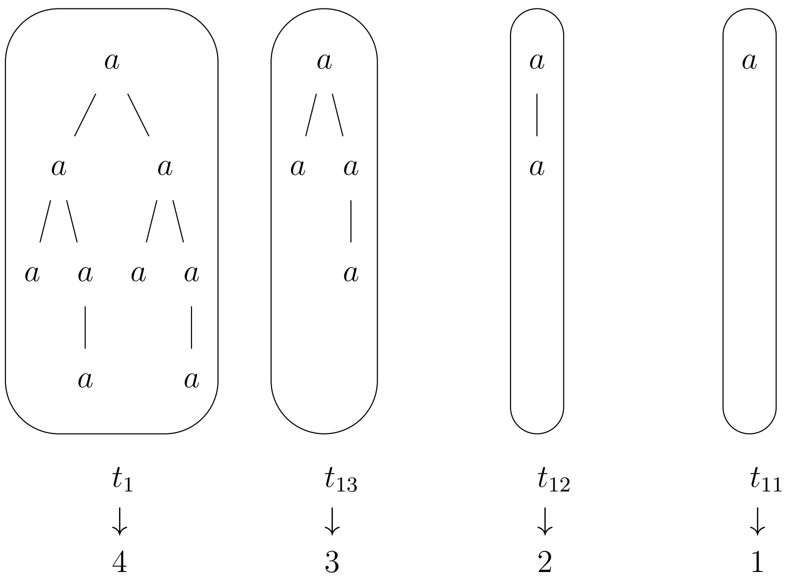


Figure 4.2: Subtree identification mapping $(\{t_1\}, \{1, 2, 3, 4\}, \mu_1)$ maps every unique subtree of tree t_1 from Example 4.1 to a unique identifier

– tree stub of subtree t_{13}, $t_{13} = aa|aa|||$, $\mu_1(t_{13}) = 3$: $(a, (1, 2))$,
– tree stub of tree t_1, $t_1 = aaa|aa|||aa|aa||||$, $\mu_1(t_1) = 4$: $(a, (3, 3))$.

Theorem 4.6 Let (T,I,μ) be a subtree identification mapping for a set of ordered trees T. Let t be any ordered subtree of any tree in set T. Let pair (r,L) be the tree stub of tree t. Tree t can be reconstructed from mapping (T,I,μ) and tree stub (r,L). At the same time, given mapping (T,I,μ), exactly one tree stub exists for every subtree from set of trees T.

The proof shows that the prefix bar notation of tree t can be reconstructed from the subtree identification mapping and its tree stub. Then it shows that a tree stub exists for every subtree of any of the trees from the set of trees T. Lastly, it proves that the tree stub can be only one for a given tree.

P r o o f. (1) Let tree t with k child subtrees be written as tree stub $(r, (\ \mu(\mathit{child\_subtree}_1), \mu(\mathit{child\_subtree}_2), \ldots))$. The prefix bar notation of t is $r\ \mathit{pref\_bar}(\mathit{child\_subtree}_1)\ \mathit{pref\_bar}(\mathit{child\_subtree}_2) \ldots \mathit{pref\_bar}(\mathit{child\_subtree}_k)\ |$. Since mapping μ assigns the same symbol only to identical trees, it is safe to use μ to rewrite the tree stub that describes tree t into $(r,(\mathit{pref\_bar}(\mathit{child\_subtree}_1), \mathit{pref\_bar}(\mathit{child\_subtree}_2), \ldots))$. This pair can then be easily transformed into $\mathit{pref\_bar}(t)$. Tree t can be reconstructed.

(2) Suppose that there is a subtree t' from set of trees T for which there exists no tree stub. Tree t' has a prefix bar notation and a root r. If the depth of tree t' is 0 and therefore $\mathit{pref\_bar}(t') = r|$, tree t' can be rewritten into a tree stub $(r, ())$, which contradicts the initial assumption and therefore this tree t' cannot exist. Suppose the depth of t' is $d + 1$. The $\mathit{pref\_bar}(t')$ can be rewritten into a pair $(r, (\mathit{pref\_bar}(\mathit{child\_subtree}_1), \mathit{pref\_bar}(\mathit{child\_subtree}_2), \ldots, \mathit{pref\_bar}(\mathit{child\_subtree}_k)))$. But $\mathit{pref\_bar}\ (\mathit{child\_subtree}_i)$ for i from 1 to k is a prefix bar notation of a child subtree of the tree t' that has depth at most d and for which there is a tree stub. This child subtree was assigned an element from set I. The pair can therefore be rewritten into $(r,(\mu(\mathit{child\_subtree}_1), \mu(\mathit{child\_subtree}_2), \ldots, \mu(\mathit{child\_subtree}_k)))$, which is a tree stub of tree t'. Tree t' therefore cannot exist.

(3) There can be only one tree stub $(r,(\mu(\mathit{child\_subtree}_1), \mu(\mathit{child\_subtree}_2), \ldots, \mu(\mathit{child\_subtree}_k)))$ for an ordered tree t. This is because the root of tree t is always the same node r, the identifier $\mu(\mathit{child\_subtree}_i)$ for i from 1 to k maps to a single value by definition and the order of the subtrees is only one for one tree t. □

Corollary 4.7 (extension of Theorem 4.6) Given a subtree identification mapping (T, I, μ), there exists a unique mapping between the set of identifiers I and the set of tree stubs of T.

P r o o f. By the previous theorem, every tree has exactly one tree stub and every tree also has exactly one identifier from the set of identifiers I. □

Example 4.8 Reconstruction of tree t_1 from Example 4.1 from the tree stubs from Example 4.5:

- $t_{11} = a|$, $\mu(t_{11}) = 1$: $(a, ()) \to a|$,
- $t_{12} = aa||$, $\mu(t_{12}) = 2$: $(a, (1)) \to (a, (a|)) \to aa||$,
- $t_{13} = aa|aa|||$, $\mu(t_{13}) = 3$: $(a, (1, 2)) \to (a, (a|, aa||)) \to aa|aa|||$,
- $t_1 = aaa|aa|||aa|aa||||$, $\mu(t_1) = 4$: $(a, (3, 3)) \to (a, (aa|aa|||, aa|aa|||)) \to aaa|aa|||aa|aa||||$.

Definition 4.9 A *general tree compression automaton* for a set of trees T - *GTCA(T)* - is a pushdown automaton $M = (Q, \mathcal{A} \cup \{|, \dashv\}, \mathcal{A} \cup I \cup \{\#\}, \delta, q_0, \#, \emptyset)$. $\mathcal{A}$ is a set of labels of the nodes of the trees from set T and I is a set of symbols. Symbol $\dashv$ is a marker symbol for the end of the input string. The automaton accepts input by an empty pushdown store. $\mathcal{A} \cap \{\#, \dashv\} = \emptyset$, $(\mathcal{A} \cup \{\#\}) \cap I = \emptyset$. A GTCA(T) accepts (by definition) exactly all subtrees of the trees in set T in the prefix bar notation.

Definition 4.10 An *initial general tree compression automaton* (*Initial GTCA*) is a $GTCA(\emptyset) = (q_0, \{|, \dashv\}, \{\#\}, \emptyset, q_0, \#, \emptyset)$.

Definition 4.11 Tree Compression Automaton - TCA
A $GTCA(T)$ is a *tree compression automaton* ($TCA(T)$) if

1. $T = \emptyset$, or
2. $T = T' \cup \{t\}$ and $GTCA(T)$ is the output of Algorithm 4.12 (TCA-construction) with input $TCA(T')$ and t.

4.2 Construction of TCA

The following algorithm describes an on-line algorithm that extends a $TCA(T)$ to create an automaton $TCA(T \cup \{t\})$. As proved later, this automaton is a $GTCA(T \cup \{t\})$.

If pushdown store P used by the following algorithm is considered as the pushdown store of automaton $TCA(T)$, then it is clear that the algorithm simulates automaton $TCA(T)$ while trying to accept input tree t. If a transition rule is missing in automaton $TCA(T)$, the algorithm extends the transition function. If automaton $TCA(T)$ does not accept a subtree that it should accept, the transition function is extended in Step 28.

A TCA constructed for a given input is shown in the following example.

Example 4.13 TCA-construction
Consider tree t_1 from Example 4.1. Given $pref\_bar(t_1)$ and an Initial $TCA(\emptyset)$, Algorithm 4.12 constructs a TCA($\{t_1\}$). As more subtrees are processed, the TCA grows progressively. Figures 4.3, 4.4 and 4.5 show how the TCA grows during construction. The resulting TCA is shown in Figure 4.6.

Theorem 4.14 The output of Algorithm 4.12 (TCA-construction) is a deterministic pushdown automaton.

Algorithm 4.12: Construction of Tree Compression Automaton

```
Name: TCA-construction
Input: A tree $t$ in prefix bar notation and an automaton $M = TCA(T)$
Output: Automaton $TCA(T \cup \{t\})$
begin
    let $M = (Q, \mathcal{A}, G, \delta, q_0, \#, \varnothing)$ be the input pushdown automaton; let $P$ be a
      pushdown store; let $q_{act}$ mark the current state;
    let the pushdown store $P = (\#)$; Let $i = 0$;
    if $\delta(q_0, |, \varepsilon) \neq (q_1, \varepsilon)$ then
        set $Q := Q \cup \{q_1\}$ and $\delta(q_0, |, \varepsilon) := (q_1, \varepsilon)$;
    end
    while $P \neq ()$ do
        while $i < |pref\_bar(t)|$ and $pref\_bar(t)[i] \neq |$ do
            $a := pref\_bar(t)[i]$;
            $\mathcal{A} := \mathcal{A} \cup \{a\}$, $G := G \cup \{a\}$, $\delta(q_0, a, \varepsilon) := (q_0, a)$; push symbol $a$ on top of
              pushdown store $P$;
            $i := i + 1$;
        end
        $q_{act} := q_1$;
        repeat
            pop a symbol $b$ from the top of pushdown store $P$;
            if $b \notin \mathcal{A}$ then
                if $\delta(q_{act}, \varepsilon, b) = (q_b, \varepsilon)$ then
                    $q_{act} := q_b$;
                else
                    create a state $q_{new}$;
                    $Q := Q \cup \{q_{new}\}$, $\delta(q_{act}, \varepsilon, b) := (q_{new}, \varepsilon)$, $q_{act} := q_{new}$;
                end
            end
        until $b \in \mathcal{A}$;
        if $\delta(q_{act}, \varepsilon, b) = (q_b, c)$ then
            push symbol $c$ on top of the pushdown store $P$;
        else
            create a new pushdown store symbol $s_{new}$; Set $G := G \cup \{s_{new}\}$,
              $\delta(q_{act}, \varepsilon, b) := (q_0, s_{new})$ and push $s_{new}$ on the pushdown store $P$.
              $\delta(q_0, \dashv, s_{new}\#) := (q_0, \varepsilon)$;
        end
        if pushd. store $P$ contains only $s\#$, $s \notin \mathcal{A}$ then set $P := ()$;
    end
    Exit and output automaton $M = TCA(T \cup \{t\})$;
end
```

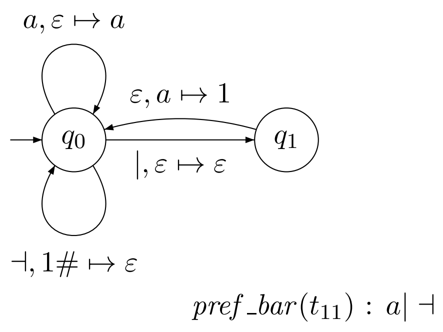

$a, \varepsilon \mapsto a$
$\varepsilon, a \mapsto 1$
q_0
q_1
$|, \varepsilon \mapsto \varepsilon$
$\dashv, 1\# \mapsto \varepsilon$
$pref\_bar(t_{11}) : a| \dashv$

Figure 4.3: TCA($\{t_{11}\}$) constructed by Algorithm 4.12

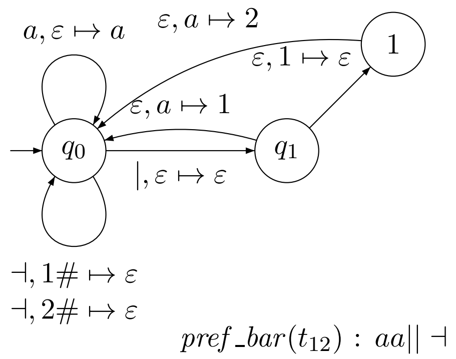

$a, \varepsilon \mapsto a$
$\varepsilon, a \mapsto 2$
$\varepsilon, 1 \mapsto \varepsilon$
1
$\varepsilon, a \mapsto 1$
q_0
q_1
$|, \varepsilon \mapsto \varepsilon$
$\dashv, 1\# \mapsto \varepsilon$
$\dashv, 2\# \mapsto \varepsilon$
$pref\_bar(t_{12}) : aa|| \dashv$

Figure 4.4: TCA($\{t_{12}\}$) constructed by Algorithm 4.12

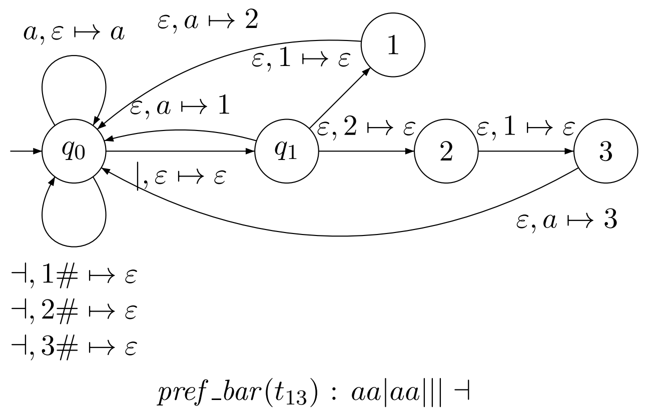
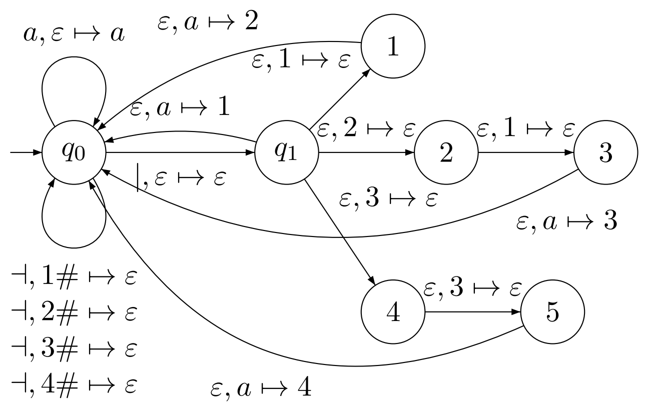

$a, \varepsilon \mapsto a$
$\varepsilon, a \mapsto 2$
$\varepsilon, 1 \mapsto \varepsilon$
1
$\varepsilon, a \mapsto 1$
q_0
q_1
$\varepsilon, 2 \mapsto \varepsilon$
2
$\varepsilon, 1 \mapsto \varepsilon$
3
$|, \varepsilon \mapsto \varepsilon$
$\varepsilon, a \mapsto 3$
$\dashv, 1\# \mapsto \varepsilon$
$\dashv, 2\# \mapsto \varepsilon$
$\dashv, 3\# \mapsto \varepsilon$
$pref\_bar(t_{13}) : aa|aa||| \dashv$

Figure 4.5: TCA($\{t_{13}\}$) constructed by Algorithm 4.12



$a, \varepsilon \mapsto a$
$\varepsilon, a \mapsto 2$
$\varepsilon, 1 \mapsto \varepsilon$
1
$\varepsilon, a \mapsto 1$
q_0
q_1
$\varepsilon, 2 \mapsto \varepsilon$
2
$\varepsilon, 1 \mapsto \varepsilon$
3
$|, \varepsilon \mapsto \varepsilon$
$\varepsilon, 3 \mapsto \varepsilon$
$\varepsilon, a \mapsto 3$
$\dashv, 1\# \mapsto \varepsilon$
$\dashv, 2\# \mapsto \varepsilon$
$\dashv, 3\# \mapsto \varepsilon$
$\dashv, 4\# \mapsto \varepsilon$
4
$\varepsilon, 3 \mapsto \varepsilon$
5
$\varepsilon, a \mapsto 4$

$pref\_bar(t_1) : aaa|aa|||aa|aa|||| \dashv$

Figure 4.6: TCA($\{t_1\}$) constructed by Algorithm 4.12

Proof. The input of Algorithm 4.12 is either an Initial GTCA, which is trivially deterministic, or an automaton that it output on a previous run. The output automaton is shown to be a deterministic pushdown automaton if one can show that it is deterministic at every step of the algorithm. Assume that the input automaton is a deterministic pushdown automaton. The output automaton M has a certain structure that follows from the algorithm:

- There are three groups of transition rules going out of the state q_0:
  1. $\delta(q_0, |, \varepsilon) = (q_1, \varepsilon)$,
  2. $\delta(q_0, a, \varepsilon) = (q_0, \varepsilon)$, $a \in (\mathcal{A} \setminus \{|, \dashv\})$,
  3. $\delta(q_0, \dashv, a\#) = (q_0, \varepsilon)$, $a \in G$.

  If the automaton M is in the state q_0, then δ is clearly a mapping for any input symbol s. The algorithm ensures that the relation δ remains a mapping by carefully checking if $\delta(q, a)$ is defined before attempting to define a value for $\delta(q, a)$.
- There are two types of transition rules going out of all states q other than q_0:
  1. $\delta(q, \varepsilon, a) = (q_0, i)$, $a \in \mathcal{A}$, $i \in G$,
  2. $\delta(q, \varepsilon, a) = (q', \varepsilon)$, $a \notin \mathcal{A}$.

  Again, if automaton M is in state q, then it is unambiguous which group of pairs from mapping δ to choose from when looking for a transition rule for an input symbol a. As before, the algorithm ensures that relation δ remains a mapping.

The trivial input automaton, Initial GTCA, is deterministic. The relation δ remains a mapping throughout the algorithm. The output automaton of the algorithm is a deterministic pushdown automaton. □

Theorem 4.15 (No cyclic configurations in a TCA) Let an automaton M be the output of Algorithm 4.12 (TCA-construction). Let it be in configuration (q, α, β). The sequence of transitions $(q, \alpha, \beta) \vdash^+ (q, \alpha, \beta)$ is not possible.

Proof. There are two types of states that automaton M can be in: q_0 and the others.

- Let automaton M be in configuration (q_0, α, β). There are no ε-transitions from this configuration. If automaton M reads a symbol from the input string, it cannot get back into configuration (q_0, α, β). Note that any input symbol $a \in \mathcal{A}$ is put on the pushdown store only if it is also read from the input string.
- Let automaton M be in configuration (q, α, β), where $q \neq q_0$. While $q \neq q_0$, every transition removes an element from the pushdown store and pushes no element back. This means that even if automaton M can get from state q back to state q by a nonempty sequence of transitions, the pushdown store contents in these two configurations will be different, unless automaton M passes through state q_0.

□

Definition 4.16 Let $\delta(q, \varepsilon, a) = (q_0, i)$ be a transition rule of an automaton $TCA(T)$. The symbol i is called a *subtree identifier.*

It is shown later in this Section that for an automaton $TCA(T)$ there exists a subtree identification mapping (T, I, μ) such that I is a set of the subtree identifiers from automaton $TCA(T)$.

Theorem 4.17 Let t be a tree and T be a set of trees. Let $M = (Q, \mathcal{A}, G, \delta, q_0, \#, \varnothing)$ be the output automaton of Algorithm 4.12 (TCA-construction) for input $TCA(T)$ and tree t. If automaton M is in the configuration $(q_0, \mathit{pref\_bar}(t)\alpha, \beta)$, then there exists a finite sequence of deterministic transitions such that $(q_0,\ \mathit{pref\_bar}(t)\alpha,\ \beta) \vdash (q_0,\ \alpha,\ i\beta)$, where i is the subtree identifier of tree t.

Proof. Automaton M is a deterministic automaton. See the proof of Theorem 4.14.

Algorithm 4.12 simulates the pushdown automaton it creates. Whenever a transition rule is missing, the algorithm first extends the transition function to enable it and then continues simulation. This proof shows that the algorithm does not extend the transition function in a way that would contradict the Theorem.

State q_0 is the only state in which automaton M reads symbols from the input string.

Let $\mathit{pref\_bar}(t) = a|$. Automaton M can only be in configuration $(q_0,\ a|\alpha,\ \beta)$, if it starts reading $a|$ from its input string. It then takes the transition $(q_0, a|\alpha, \beta) \vdash (q_0, |\alpha, a\beta)$. Since automaton M is deterministic, Algorithm 4.12 has to take this transition while simulating TCA for input string $a|$. Symbol a is now on the top of the pushdown store. A bar is the next input symbol, which forces Algorithm 4.12 to take the transition $(q_0, |\alpha, a\beta) \vdash (q_1, \alpha, a\beta)$.

While in the configuration (q, α, γ), $q \neq q_0$, $\gamma = \gamma' a\beta$, $\gamma' \in G*$, automaton M has to behave deterministically independent of the input, deciding the transitions only based on the symbols from γ. The only transitions that lead automaton M back into the state q_0 are the transitions that read a symbol a, $a \in \mathcal{A}$, from the top of the pushdown store. The sequence of transitions $(q_1, \alpha, \gamma) \dashv^{+} (q_x, \alpha, a\beta)$ is thus completely determined by γ down to the first symbol $a \in \mathcal{A}$. This sequence of transitions is finite because there can be no "cycle" in the sequence of configurations (see the proof of Theorem 4.15).

Automaton M makes transition from configuration to configuration until it pops a symbol $a, a \in \mathcal{A}$ from the top of the pushdown store. At that moment automaton M has performed a sequence of transitions that uniquely identifies subtree $a|$. The automaton M assigns a subtree identifier i to $a|$ and pushes i on the pushdown store. Automaton M performed the following transition: $(q_1, \alpha, a\beta) \vdash (q_0, \alpha, i\beta)$.

Let $pref\_bar(t) = a\ pref\_bar(t_1) \dots pref\_bar(t_k)|$. Let automaton M be in the configuration $(q_0, a\ pref\_bar(t_1) \dots pref\_bar(t_k)|\alpha, \beta)$ when it starts reading symbols from the input string. After pushing symbol a on top of the pushdown store, it will process $pref\_bar(t_1)$ from the input string, eventually making a transition to the configuration $(q_0, pref\_bar(t_2) \dots pref\_bar(t_k)|\alpha, i_1 a\beta)$. This is repeated for the remaining subtrees in the input string until automaton M makes a transition to the configuration $(q_0, |\alpha, i_k \dots i_1 a\beta)$. At this moment, the same reasoning as in the previous paragraph can be applied. There is exactly one finite sequence of transitions that leads automaton M from the configuration $(q_0, |\alpha, i_k \dots i_1 a\beta)$ to the configuration $(q_0, \alpha, i\beta)$. □

Theorem 4.18 Algorithm 4.12 (TCA-construction) constructs a $GTCA(T \cup \{t\})$ for a given $TCA(T)$ and a tree t.

The proof consists of two parts. In the first part, it is shown that the output of Algorithm 4.12 still recognises all subtrees of the trees from T.

In the second part of the proof, it is first proved that the constructed automaton accepts all subtrees of tree t. Then it is shown that the constructed automaton does not accept anything else than subtrees from $T \cup \{t\}$.

Proof. Firstly, the input automaton $TCA(T)$ and the output automaton as well are pushdown automata, which directly follows from the definition of the TCA. Throughout Algorithm 4.12 nothing is deleted from the input $TCA(T)$. The $TCA(T)$ is modified only through additions to its sets $Q, \mathcal{A}, G$ and extension of the transition function δ. Since nothing is deleted from the transition function, the language that the $GTCA(T \cup T')$ pushdown automaton accepts must contain the language that $TCA(T)$ accepts.

Secondly, let $pref\_bar(t) = a|$. Let $T' = \{t\}$. Let the input of Algorithm 4.12 be a $TCA(T)$ automaton M and set T'. Let the contents of the pushdown store of automaton M be β. The algorithm starts simulating automaton M for input $a|\alpha$. In this case $\beta = \#, \alpha = \dashv$. The algorithm puts the node label a on the pushdown store (simulating either an existing transition rule or a newly created transition rule $\delta(q_0, a, \varepsilon) = (q_0, a)$). It then reads a bar from the input and has to transition to state q_1. The tree that was read is present on pushdown store P. On the top of pushdown store P is root a of a subtree, $a \in \mathcal{A}$. The automaton then performs a sequence of transitions that ends in state q_0 and stores an identifier i for the just read subtree on the pushdown store. The content of the input string is now α, the pushdown store holds $i\beta$ on the top. The transition function is extended in automaton M to accept tree t by emptying the pushdown store if it is the only tree on input. If the depth of tree t is zero, automaton M is constructed to accept it.

Any tree or subtree on input of Algorithm 4.12 should be accepted by the constructed TCA. For input starting with $\mathit{pref\_bar}(t)$, the automaton makes a sequence of transitions from the configuration $(q_0,\mathit{pref\_bar}(t)\alpha,\beta)$ to the configuration $(q_0,\alpha,i\beta)$. The transition function is extended if necessary: $\delta(q_0,\dashv,i\#)=(q_0,\varepsilon)$. Automaton M makes a transition using δ to accept a tree t if $\alpha=\dashv$.

Assume that every tree t' of depth at most k that is a subtree of a tree t of depth $k+1$ is accepted by a $TCA(T \cup \{t\})$ thanks to Algorithm 4.12. This assumption implies that the TCA will have the subtree identifier of tree t' on top of its pushdown store after a recognised tree t' of depth at most k is read by it from the input string.

Let t denote a tree of depth $k+1$ with n child subtrees that is put on the input of $TCA(T)$ by Algorithm 4.12. When a bar symbol b that corresponds to the root symbol a of tree t is reached while reading the $\mathit{pref\_bar}(t)$, the subtree identifiers of child subtrees $\mathit{subtree}_j$ of tree t for $j=1$ to $j=n$ of depth at most k present between symbol a and its bar b are already present on the pushdown store in the reverse order of appearance in the original tree, in the form of subtree identifiers. This follows from Theorem 4.17.

The pushdown store looks like this: $P=\alpha_n\alpha_{n-1}...\alpha_1 a$, $a \in \mathcal{A}$, $\alpha_j \in Q$ for $j=n$ to $j=1$, i.e. on and just below the top of the pushdown store are subtree identifiers of all the subtrees that were sequentially identified after symbol a was read but before bar symbol b was reached. The subtree identifiers are stored on the pushdown store in reverse order of the subtrees of tree t.

Algorithm 4.12 creates a sequence of new states and extends transition function δ for the sequence of subtree identifiers, if necessary. If state q is the last state of this sequence, Algorithm 4.12 ensures that there will be a transition rule $\delta(q,\varepsilon,a)=(q_0,m)$, $a \in \mathcal{A}$, with a unique m. As there exists a transition rule that empties the pushdown store in the case that only $m\#$ is on it and there is nothing left on the input, the automaton M will accept tree t.

It remains to show that $TCA(T)$ does not accept anything else than the subtrees of trees from set T.

- Identifier i that is pushed on the pushdown store is always the same for the same input tree t - that is based on the determinism of the automaton and on the fact that if the automaton accepts tree t, then the sequence of transitions from the configuration $(q_0,\mathit{pref\_bar}(t)\alpha,\beta)$ to the configuration $(q_0,\alpha,i\beta)$ must be possible within automaton M.
- Identifier i is also unique for tree t. That is, it is not possible for automaton M to get from configuration $(q_0,\ \mathit{pref\_bar}(t')\alpha,\ \beta)$ to configuration $(q_0,\ \alpha,\ i\beta)$ if $\mathit{pref\_bar}(t) \neq \mathit{pref\_bar}(t')$. It is easy to see that the in-degree (number of transitions leading to the state) of every state except state q_0 of automaton M is 1. Whenever a transition rule is missing in automaton M (except for a transition that should lead to state q_0), Algorithm 4.12 extends the transition function. The algorithm also extends the transition function $\delta(q,\varepsilon,a \in \mathcal{A})=(q_0,m)$, if necessary, using a unique m. Therefore for two different $\delta(q_1,\varepsilon,a)=(q_0,m), \delta(q_2,\varepsilon,b)=(q_0,n)$, $a,b \in \mathcal{A}$, it holds $m \neq n$. Identifier i is unique for tree t.

– Algorithm 4.12 assigns only as many subtree identifiers as necessary - based on the number of unique subtrees in input tree t that were not present in set T. Therefore if a tree t' is present in the input string of automaton M constructed by Algorithm 4.12 and t' is not one of the subtrees of $T \cup \{t\}$, then after automaton M reads tree t' from the input string and makes a transition out from state q_0, it cannot get back into state q_0 and therefore cannot accept this different tree t'.

Automaton M is a $GTCA(T \cup \{t\})$. □

Corollary 4.19 A pushdown automaton $TCA(T)$, where T is a set of trees, is a deterministic pushdown automaton.

P r o o f. The proof follows directly from the fact that the output of Algorithm 4.12 (TCA-construction) is a deterministic pushdown automaton. □

Theorem 4.20 Let T be a set of trees and let t be a subtree of any of the trees in T. Let i be the subtree identifier of t. Let $M = (Q, \mathcal{A} \cup \{|, \dashv\}, \mathcal{A} \cup I \cup \{\#\}, \delta, q_0, \#, \emptyset)$ be a $TCA(T)$ and there exists a subtree identification mapping (T, I, μ). If the transition function δ contains the values $\delta(q_0, |, \varepsilon) = (q_1, \varepsilon)$, $\delta(q_1, \varepsilon, C) = (q_2, \varepsilon)$, …, $\delta(q_{k-1}, \varepsilon, X) = (q_k, \varepsilon)$, $\delta(q_k, \varepsilon, r) = (q_0, i)$, then $(r, (X, \ldots, C))$ is a tree stub of tree t.

P r o o f. The set of pushdown store symbols G of automaton M consists of $\mathcal{A} \cup I \cup \{\#\}$, where every element from I is a subtree identifier of some subtree of some tree from set T. As shown in the proof to the previous Theorem, every unique tree is assigned a different element from set I. Also, if two subtrees are the same, automaton M will assign them the same element from set I. This implies that automaton M defines a bijective mapping μ between the set of subtrees of the trees in set T and set I. That means that there exists a subtree identification mapping (T,I,μ).

Let t be a subtree of any of the trees in set T. There must exist the following unique sequence of transitions between configurations of automaton M: $(q_0, pref\_bar(t), \alpha) \vdash^+ (q_0, \varepsilon, i\alpha)$. Symbol i is then the subtree identifier of tree t. It is unique because automaton M is deterministic.

Let $pref\_bar(t) = r pref\_bar(t_1) \ldots pref\_bar(t_k)|$. There must exist a transition $(q_0, pref\_bar(t), \alpha) \vdash (q_0, pref\_bar(t_1) \ \ldots \ pref\_bar(t_k)|, r\alpha)$. Then if i_1,…,i_k are the subtree identifiers of t_1,…,t_k respectively, there must exist a sequence of transitions $(q_0, pref\_bar(t), \alpha) \vdash (q_0, |, i_k \ldots i_1 r\alpha)$. Since every subtree identifier is an element of the I and there is a bijective mapping between set I and the subtrees of the trees in the T, $(r, (i_1, \ldots, i_k))$ is a tree stub of tree t, given the subtree identification mapping (T,I,μ). □

4.2.1 Size of the output of Algorithm 4.12 (TCA-construction)

Let automaton M be a $TCA(T)$, where T is a set of trees. Let t be a tree with n nodes. When Algorithm 4.12 is executed with tree t and automaton M as input, it may add new states, transitions and pushdown store symbols to automaton M. This Subsection computes the maximum and minimum number of states, transitions and pushdown store symbols that can be added to automaton M by the algorithm.

There are two cycles in Algorithm 4.12. It is first shown how many times each cycle is executed, and then the size of the output is established.

The first cycle (step 8 of Algorithm 4.12) reads symbols from the input string. A symbol $a \neq |$ is read and pushed on the pushdown store n times in total. Every time this happens the transition function is extended and one pushdown store symbol can be added to automaton M, adding at maximum n pushdown store symbols to the pushdown store alphabet of automaton M.

Symbol $a = |$ is read n times as well. The execution then enters the second cycle that starts with Step 14. In total, the first cycle is executed $2n$ times.

The second cycle is the pushdown store contents processing cycle at Steps 14 through 25 in the algorithm. It is reached n times from the first cycle, whenever a bar is read.

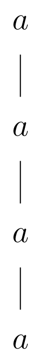
The execution of Algorithm 4.12 returns from the second cycle to the first cycle every time that a symbol from $\mathcal{A}$ is present on top of the pushdown store. If and only if the execution returns to the first cycle, the algorithm pushes a symbol $b \notin \mathcal{A}$ onto the pushdown store and can add symbol b to the pushdown store alphabet. Execution returns to the first cycle n times.

Exactly n symbols $a \in \mathcal{A}$ and n symbols $b \notin \mathcal{A}$ are pushed onto the pushdown store in total. At every execution of the body of the second cycle one symbol $b \notin \mathcal{A}$ is popped from the pushdown store. Algorithm 4.12 pops one symbol $b \notin \mathcal{A}$ and one symbol # from the pushdown store before exiting. Only $2n - 1$ symbols can be and are popped throughout the execution of the algorithm. Therefore the body of the second cycle is executed $2n - 1$ times.

Theorem 4.21 Let t be a tree with n nodes. Let M be the output TCA of the Algorithm 4.12 (TCA-construction) for input consisting of tree t and an Initial GTCA. TCA M has at most $n + 1$ states, $2n + 1$ pushdown store symbols and the number of transition rules is $4n$.

P r o o f. A new state can be added to automaton M in the body of the second cycle only when a symbol $b \notin \mathcal{A}$ is on top of the pushdown store. That happens $n - 1$ times during the execution of Algorithm 4.12 (the nth time that symbol b is on top of the pushdown store, the whole tree is accepted and no state is created). Only two states can be added to automaton M outside the body of the second cycle (states q_0 and q_1). This means that at maximum $n + 1$ states can be added to automaton M by the algorithm.

The pushdown store alphabet of an Initial GTCA contains one symbol, #. At maximum n symbols $a \in \mathcal{A}$ can be added to the pushdown store alphabet. Symbols $b \notin \mathcal{A}$ can be



$$pref\_bar(t_3) = aaaa||||$$

Figure 4.7: Tree t_3 and its prefix bar notation from Example 4.22

added to the pushdown store alphabet every time a symbol $a \in \mathcal{A}$ is popped from the pushdown store. This happens n times. Therefore at maximum n symbols $b \notin \mathcal{A}$ can be added to the pushdown store alphabet. In total, the pushdown store alphabet has at most $2n + 1$ symbols.

The transition function is extended $\delta(q_0, a, \varepsilon) = (q_0, a)$ for every symbol $a \in \mathcal{A}$ that is read from the input. This can be at most n times. The transition function is extended $\delta(q_0, \varepsilon, b\#) = (q_0, \varepsilon)$ for every pushdown store symbol $b \notin \mathcal{A}$. This can be again at most n times. One extension of δ can be added from state q_0 to state q_1. One extension of δ can be added by Step 28 for every symbol $a \in \mathcal{A}$ on top of the pushdown store. This is again at most n extensions. One extension of δ is added by the second cycle every time a new state is added. This is at most $n - 1$ times. In total, automaton M will have at most $4n$ transition rules . □

Trivially, the algorithm adds no new states, pushdown store symbols and extensions of δ to the $TCA(T)$ if input tree t is a subtree of any of the trees in set T. Note that an implementation of the TCA can omit the storing of transition rules of type $\delta(q_0, a, \varepsilon) = (q_0, a)$, $a \in \mathcal{A}$, and $\delta(q_0, \varepsilon, b\#) = (q_0, \varepsilon)$, $b \notin \mathcal{A}$, because these transition rules exist for all a, b respectively. In such case only $2n$ elements of the transition function δ need to be stored.

The case of maximal output size is encountered when there are no subtree repeats in the set of input trees T.

Example 4.22 Figure 4.7 shows a tree t_3 for which a TCA provides the worst compression it is capable of. This tree has no repeating subtrees.

Theorem 4.23 Let t be a tree with n nodes. Let an automaton M be the output $TCA(\{t\})$ of Algorithm 4.12 (TCA-construction) with input consisting of tree t in its prefix bar notation and an Initial GTCA. If $n = 3^d$, where d is the depth of tree t, $TCA(\{t\})$ M can have only $2 * log_3(n) + 2$ states, $log_3(n) + 2$ pushdown store symbols and

$4 * log_3(n) + 4$ transition rules. In general, automaton M cannot have less than $\lceil log_6(n) \rceil + 2$ states, $\lceil log_6(n) \rceil + 2$ pushdown store symbols and number of transition rules smaller than $3\lceil log_6(n) \rceil + 4$ δ.

Proof. Let t be a tree for which the ratio $r =$ (number of states plus number of transition rules in $TCA(\{t\})$)/(number of nodes in t) is minimal.

If two subtrees of tree t have the same depth, they have to be identical. If they were not, then one could imagine a tree t' in which the occurrences of the smaller of the two subtrees are replaced by the greater one. $TCA(\{t'\})$ would not need to create states and extend δ, which would be needed to encode the smaller of the two subtrees while encoding a tree with at least as many nodes as there are in tree t.

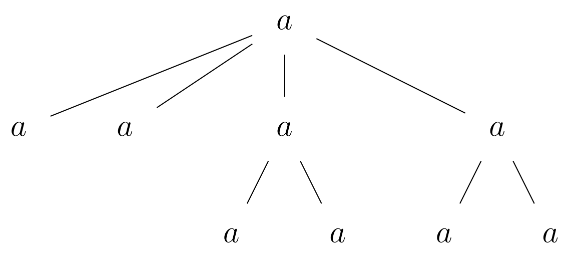
Let t_{k+1} be a subtree of depth $k+1$. As child subtrees, this subtree will have all the child subtrees of subtree t_k of depth k. The child subtrees will be in the same order as in subtree t_k. Such a tree requires no new states and no extension of δ in automaton M to encode the child subtrees copied from child subtree t_k. The subtree t_{k+1} will also have a number n_k of copies of subtree t_k as its leftmost child subtrees. To encode n_k child subtrees t_k in subtree t_{k+1}, the $TCA(\{t\})$ requires n_k new states and n_k δ extensions. To encode the root of subtree t_{k+1} and allow acceptance of subtree t_{k+1}, the $TCA(\{t\})$ needs 2 more δ extensions.

Let tree t have depth d. Let n_k be the number of subtrees of depth k in the subtree of depth $k+1$. The tree t has $(n_{d-1}+1)(n_{d-2}+1)\dots(n_0+1)$ nodes. $TCA(\{t\})$ will have $4 + 2d + (n_{d-1} + n_{d-2} + ... + n_0)$ transition rules, $2 + (n_{d-1} + n_{d-2} + ... + n_0)$ states and $2 + d$ pushdown store symbols.

Number n_k cannot be higher than 5. If there were 6 child subtrees of depth k in subtree s, the subtree s would have $7 * nodes(k)$ nodes, where $nodes(k)$ is the number of nodes in the subtree of depth k. The encoding of s would require 6+2 additional δ extensions, 6 additional states and one additional pushdown store symbol in a TCA that already accepts the subtree of depth k. One could imagine a subtree s_1 with 2 child subtrees of depth k and a subtree s_2 with 2 child subtrees of depth $k+1$. s_2 would have $3 * 3 * (nodes(k))$ nodes, but s_1 and s_2 would require only 4+4 additional δ extensions, 4 additional states and 2 additional pushdown store symbols in the TCA. Therefore s cannot be present in tree t. The same reasoning can be applied if s has 7 and 8 child subtrees of depth k.

If s has 3^2 child subtrees of depth k and therefore has $10 * (nodes(k))$ nodes, one would imagine a subtree s_3 with two child subtrees of depth $k+2$, which again requires less space in the TCA and encodes the tree with more nodes, $27 * (nodes(k))$. Subtree s has more than $27 * (nodes(k))$ nodes if it has at least 3^3 child subtrees of depth k. Adding more $(3^3, 3^4, \dots)$ child subtrees to s allows encoding of trees with $3^3, 3^4, \dots$ times more nodes while requiring $3^3, 3^4, \dots$ times more space in TCA. Adding subtrees s_i with increasing depth $(i = k+3, i = k+4, \dots)$ allows encoding of trees with as many nodes while requiring only $3 * 2, 4 * 2, \dots$ times more space in the TCA.

Ratio r is equal to $8 + 3d + 2(n_{d-1} + n_{d-2} + ... + n_0)/(n_{d-1}+1)(n_{d-2}+1)\dots(n_0+1)$. Ratio r is minimal for n_k with values from $\{1,2,3,4,5\}$, where k ranges from 1 to $d-1$. The minimal value of ratio r therefore cannot be higher than $(8 + 3d + 2 * (d * 5))/2^d$. In



$$pref\_bar(t_2) = aa|a|aa|a||aa|a|||$$

Figure 4.8: Tree t_2 and its prefix bar notation from Example 4.24

fact, it cannot be higher than $(8+3d+2*(d*2))/3^d$, the value achieved if $n_k = 2$. Ratio r also cannot be lower than $(8+3d+2*(d))/6^d$. The exact lower bound for r is not known. □

The best compression ratio is achieved for example for Fibonacci trees and full k-ary trees. A Fibonacci tree of order n is a binary tree with the left subtree of order $n-1$ and the right subtree of order $n-2$. A Fibonacci tree of order 0 has no nodes. A Fibonacci tree of order 1 has 1 node [43].

Example 4.24 Figure 4.8 shows a tree t_2 for which a TCA provides the best compression it is capable of. This tree has a large number of repeating subtrees.

4.3 Tree decompression from TCA

The output of Algorithm 4.12 (TCA-construction), a $TCA(T)$, will be shown as suitable for compression of trees that contain repeating subtrees. The compression ratio ranges from linear to logarithmic. The worst case is encountered if the compressed tree contains no repeating subtrees, as in Example 4.22. The best case is a tree that has all subtrees of the same depth identical, while keeping number of child subtrees between 2 and 6. Such trees are found in the proof to Theorem 4.23. Example 4.24 shows such a tree.

The decompression algorithm reconstructs tree t from $TCA(T)$ if t is a subtree of any of the trees in set T. The main idea of the decompression algorithm is to transform $TCA(T)$ into a straight-line grammar [11] that generates exactly one string, tree t. The decompression algorithm needs two things on the input: $TCA(T)$ and the subtree identifier i that was assigned to tree t by Algorithm 4.12 (TCA-construction).

The subtree identifier i of tree t has to be remembered if set $T \neq \{t\}$. Else it can be found as the only subtree identifier for which $\delta(q, \varepsilon, i) = \varnothing$ for all states $q \in Q$.

Example 4.26 Consider tree t_1 from Example 4.1. The $TCA(\{t_1\})$ constructed for this tree is shown in Figure 4.6. The grammar that generates $pref\_bar(t_1)$, constructed by Algorithm 4.25, is $R_{t1} = (\{1,2,3,4\}, \{a, |\}, P_{t1}, 4)$, where

Algorithm 4.25: Decompression of a tree from TCA

```
Name: TCA-decompression
Input: Automaton $TCA(T)$. Subtree identifier $i$ assigned to a tree $t$ by
       Algorithm 4.12 during construction of $TCA(T)$.
Output: Tree $t$ in prefix bar notation.
1  begin
2      let $M = (Q, \mathcal{A}, G, \delta, q_0, \#, \varnothing)$ be the $TCA(T)$; let $q_{act}$ be a marked state;
3      create a grammar $R = (N, \mathcal{A} \cup \{|\}, P, S)$, where $N = G \setminus \mathcal{A} \setminus \{\#\}$, $S = i$ and $P$
        is an empty set of rules;
4      reverse the transition function in $TCA(T)$, that is, replace every
        $(p, v, \alpha) \rightarrow (q, \beta)$ by $(q, v, \alpha) \rightarrow (p, \beta)$;
5      foreach $x \in \mathcal{A}, q_y \in Q, C \in G$ such that $\delta(q_0, \varepsilon, x) = (q_y, C)$ do
6          create a rule $r = C \rightarrow x$ and set $q_{act} := q_y$;
7          while exists $D \in G, q_z \in Q$ such that $\delta(q_{act}, \varepsilon, D) = (q_z, \varepsilon)$ do
8              append $D$ to the right-hand side of rule $r$ and set $q_{act} := q_z$;
9              If $q_{act} = q_0$, set the rules $P := P \cup \{r\}$ and continue Step 5 for the next
                element from transition function $\delta$;
10         end
11     end
12     For output, generate any string of language $L(R)$ (this language will be shown to
        contain exactly one string, $pref\_bar(t)$);
13 end
```

$$
\begin{aligned}
P_{t1} = \{\ 4 &\rightarrow a33|,\\
3 &\rightarrow a12|,\\
2 &\rightarrow a1|,\\
1 &\rightarrow a|\ \}.
\end{aligned}
$$

Compare this straight-line grammar with the tree stubs for tree t_1 from Example 4.5.

Theorem 4.27 Let t be a tree. Let M be a $TCA(T)$ such that tree t is a subtree of one of the trees in set T. Let i be a subtree identifier assigned to tree t by automaton M. Let w be the output of Algorithm 4.25 (TCA-decompression) whose input was automaton M and subtree identifier i. Then $w = \mathit{pref\_bar}(t)$.

Proof. Let u be any subtree of any of the trees in set T. Let r_u be the root of tree u. Let i_u be the subtree identifier of subtree u. Let $\mathit{pref\_bar}(u) = r\ \mathit{pref\_bar}(s_1) \dots \mathit{pref\_bar}(s_k)|$ and let $i_1, \dots, i_k$ be the subtree identifiers of subtrees $s_1, \dots, s_k$, respectively. Then for input $< \mathit{pref\_bar}(u) > \alpha$, automaton M must be able to take a sequence of transitions into a configuration $(q_0, |\alpha, i_k \dots i_1 r)$. From Algorithm 4.12 (TCA-construction), it holds that
$(q_0, |\alpha, i_k \dots i_1 r) \vdash (q_1, \alpha, i_k \dots i_1 r) \vdash (q_2, \alpha, i_{k-1} \dots i_1 r) \vdash \dots \vdash (q_0, \alpha, i_u)$
and all states $q_i, q_j, i \neq j$ are pairwise different.

Let M' be a pushdown automaton equivalent to automaton M that has its transition function δ reversed. Directly following from Algorithm 4.12, $\delta(q_0, \varepsilon, x) = \{(q_y, C)\}$ exists only if C is a subtree identifier of a subtree of any of the trees in T. Also based on Algorithm 4.12, for every state other than q_0 in automaton M' there is exactly one outgoing transition rule $\delta(q, \varepsilon, x) = \{(q_y, C)\}$. Based on the previous paragraph, it must then hold that $\delta(q_0, \varepsilon, r_u) = (q_k, i_u), \delta(q_k, \varepsilon, i_1) = (q_{k-1}, \varepsilon), \dots, \delta(q_2, \varepsilon, i_k) = (q_1, \varepsilon), \delta(q_1, |, \varepsilon) = (q_0, \varepsilon)$.

The Algorithm 4.25 thus constructs the rules r of grammar R in the form $r = i_u \rightarrow r_u i_1 \dots i_k|$ for every subtree u. If u is a subtree of depth 0, then the right-hand side of rule r is the prefix bar notation of u. If u is a subtree of depth $d+1$, then again the right-hand side of rule r is the prefix bar notation of u if the non-terminals $i_1, \dots, i_k$ are "transitively" rewritten to their right-hand sides. That is exactly how grammar R generates its language if the initial symbol is set to be the non-terminal i_u.

□

4.4 Time and space complexity of compression & decompression

Lemma 4.28 Let n be the number of nodes of a tree that is an input to Algorithm 4.12 (TCA-construction). When a hash map is used for the storage of the transition function δ, Algorithm 4.12 requires $\mathcal{O}(2n)$ time and $\mathcal{O}(2n)$ space.

Proof. The time and space complexity of the Algorithm 4.12 with input consisting of a $TCA(T)$ and a tree t with n nodes depends on the implementation of the lookup of transition function δ.

If the algorithm is provided with a zeroed space of size $(2n+1) \times |Q|$, a lookup table can be created for transition function δ. Finding an appropriate transition in such table takes constant time. The space taken by transition function δ is $(2n + 1) \times |Q|$. Alternatively, a hash map can be used for storing the transition function δ. In fact, a hash map was used for this purpose in a working implementation of TCA [52].

If the algorithm does not construct a lookup table or a hash map for transition function δ, then a searching algorithm must be used for finding the appropriate transition rule. The maximum size of a set of outgoing transition rules from a state is $n + 1$ in the case of state q_0. The transition rule lookup time is therefore at worst $log_2(n + 1)$. If each element of transition function δ takes up space $log2(n + 1)$, space taken by transition function δ is at worst $4n * log_2(n + 1)$ or just $2n * log_2(n + 1)$ when the optimization mentioned below the proof of Theorem 4.21 is used.

The total time and space requirements of Algorithm 4.12 depend directly on the implementation of transition function δ. The Subsection 4.2.1 that computes the size of the output TCA showed that there were two cycles in the algorithm. The first cycle is run n times, and the second cycle is also run n times.

Every run of the body of the first cycle has to find $\delta(q_0, a, \varepsilon)$. If a lookup table or a hash map is available, then the body of the first cycle is executed in constant time. If a lookup table is not available, then one run of the body of the first cycle requires $c+log_{2+1}n$ time, where c is a constant.

One run of the body of the second cycle has to find $\delta(q_{act}, \varepsilon, b)$ and can add a new transition rule to transition function δ. The rest of the body executes in constant time and can require constant space. Again depending on the implementation of transition function δ, the execution of the body of the second cycle can take either constant or $c + log_{2+1}n$ time.

Depending on the implementation of transition function δ (hash map or array), Algorithm 4.12 (TCA-construction) can either require at worst $c_1 * n = \mathcal{O}(n)$ time and $2n * log_2(n + 1)$ space or at worst $c_2 * 2n * log_2(n + 1)$ time and $2n * log_2(n + 1)$ space, respectively. c_1, c_2 are constants of the algorithm. For reasonable sizes of n (less than 2^{64}), the $log_2 n + 1$ part of the space complexity is reduced to 1 on modern 64-bit computer hardware and the space complexity is in $\mathcal{O}(2n)$. □

The implementation of TCA [52] used a hash map implementation provided by the C++ standard library (std::unordered_map). This map has a guaranteed constant average insert, erase and retrieve time [39].

Lemma 4.29 Let n be the number of nodes of a tree that is the output of Algorithm 4.25 (TCA-decompression). When a hash map is used for the storage of the transition function δ, Algorithm 4.25 requires $\mathcal{O}(5n)$ time and $\mathcal{O}(3n)$ space.

Proof. If transition function δ is implemented using a lookup table or a hash map, the δ lookup time is constant and the transition rule reversal takes time $2n$ (same optimization of the transition function is assumed as in the previous Lemma). If δ is not implemented through a lookup table or a hash map, then the transition lookup time is at worst $log_2(n+1)$ and reversal of the transition rules will take time at worst $(2n * log_2(n+1))$. For the rest of the analysis, we assume that a hash map is used for the storage of the reversed transition function.

All operations in Step 6 and Steps 8 through 9 of the algorithm can be made within constant time c_1, , $c_1 \geq 1$. In total, Step 6 can be performed at most n times and Steps 8 through 9 can be performed at most n times. This in total requires at most $c_1 * 2 * n$ time.

The grammar $R = (N, \mathcal{A} \cup \{|\}, P, S)$ will generate its language in time less than or equal to $c_2 * n$, $c_2 \geq 1$. The size of the generated language will be n.

Let $|TCA|$ be the size of the TCA on input. Algorithm 4.25 requires space $|TCA| + |N| + n = \mathcal{O}(3n)$. The algorithm requires time $2n + c_1 * 2 * n + c_2 * n = \mathcal{O}(5n)$. $|N|$ is guaranteed to be less than or equal to n, because each non-terminal denotes a subtree. □

4.4.1 Compression and decompression conclusion

When TCA is transformed into a grammar using Algorithm 4.12 (TCA-decompression), it is obvious that the compression method that uses Algorithm 4.12 (TCA-construction) is similar to a technique for grammar compression of trees [11].

The proposed compression algorithm for trees offers a good compression ratio for trees with repeating sub-patterns. It does not achieve such a good compression ratio as the comparable LZ methods [71, 65], but exchanges this drawback for an output that is easy to work with if one requires for example to search for a pattern in the compressed tree.

4.5 TCA as an index of a tree

It is important to note at this moment that the $TCA(\{t\})$ is quite naturally an index of tree t. If hashing is used for storing the transition function δ, deciding whether any tree u is a subtree of tree t is an operation that takes time at most $|u|$.

The searching algorithm takes $TCA(\{t\})$ and u as input and then executes Algorithm 4.12 (TCA-construction). If at any step Algorithm 4.12 tries to add a new state to its set of states (Step 21) or to add a new pushdown store symbol to its set of pushdown store symbols (Step 28), tree u does not occur in t. Otherwise it does occur and its subtree identifier is the last symbol that is pushed on the pushdown store before the exit of Algorithm 4.12.

4.6 TCA as a matcher

A matcher is a structure that allows to search for occurences of trees from a given set of trees within any provided tree. TCA can be used as a matcher for the given set of trees $\{t_1, t_2, \ldots, t_k\}$. A TCA used as a matcher for trees $t_1, t_2, \ldots, t_k$ is simply a $TCA(\{t_1, t_2, \ldots, t_k\})$. It is constructed using Algorithm 4.12 (TCA-construction). The user of the matcher TCA must note the subtree identifiers $id_1, id_2, \ldots, id_k$ of the trees $t_1, t_2, \ldots, t_k$, respectively.

The matching algorithm is a slightly modified TCA-construction algorithm. For clarity, the algorithm is presented in Algorithm 4.30 (TCA-matching). The difference from the TCA-construction algorithm lies in the usage of the Z_{np} symbol, which stands for "non-matching subtree" and in the reporting of matching subtrees.

Algorithm 4.30 takes a tree t and $TCA(\{t_1, t_2, \ldots, t_k\})$ as input. For every symbol read from $pref\_bar(t)$, the matching algorithm increases an internal counter of position i. Whenever a subtree identifier that corresponds to a tree in $\{t_1, t_2, \ldots, t_k\}$ is pushed on the pushdown store, the algorithm outputs the value of the internal counter of position i inside input and the corresponding subtree identifier.

The matching algorithm needs to have a few special improvements so the input TCA is not changed. A special "not-present" state q_{np} is used, a special "not-present" pushdown-store symbol Z_{np} is used and a special "not-present" alphabet symbol a_{np} is used. If at any step the original matching algorithm tries to add a new state to its set of states (Step 21) or to enter this state, or to add a new pushdown store symbol (alphabet symbol) to its set of pushdown store symbols (alphabet symbols) (Step 28) or to use it, the modified matching algorithm uses its "not-present" state and "not-present" pushdown store symbol (alphabet symbol), respectively. This way the input TCA is not modified and all matches are reported.

Example 4.31 Consider tree t_2 from Example 4.24. Consider TCA($\{t_1\}$) from Example 4.13. Assume that we want to report matches of subtrees with identifiers 1,2,3 and 4 in tree t_2.

With tree t_2 and TCA($\{t_1\}$) on input, Algorithm 4.30 reports six matches of subtrees with identifier 1 in tree t_2 (essentially, only the 6 leafs are the subtrees that are common to both trees t_1 and t_2).

Lemma 4.32 Let $t, t_1, t_2, \ldots, t_k$ be trees. Let M be a $TCA(\{t_1, t_2, \ldots, t_k\})$. Let $id_1, id_2, \ldots, id_k$ be the subtree identifiers of trees $t_1, t_2, \ldots, t_k$, respectively. Algorithm 4.30 (TCA-matching) reports positions and subtree identifiers of all subtrees of t that match any of trees in $\{t_1, t_2, \ldots, t_k\}$.

Proof. Algorithm 4.30 is a modified Algorithm 4.12 (TCA-construction). Algorithm 4.12 identifies all subtrees in tree t that match any of trees in $\{t_1, t_2, \ldots, t_k\}$ after Step 25 of Algorithm 4.12. Algorithm 4.30 reports these matching subtrees at Step 27. At this step, the value of counter i is equal to the position of the end of the prefix bar notation of

Algorithm 4.30: Subtree matching using TCA

Name: TCA-matching

Input: Automaton $TCA(\{t_1, t_2, \ldots, t_k\})$. Subtree identifiers id_i assigned to tree t_i for all i, $0 < i \leq k$, by Algorithm 4.12 during construction of $TCA(\{t_1, t_2, \ldots, t_k\})$. A subject tree t.

Output: Positions of matched subtrees in $pref\_bar(t)$ and the respective subtree identifiers.

```
begin
    let $M = (Q, \mathcal{A}, G, \delta, q_0, \#, \varnothing)$ be the $TCA(\{t_1, t_2, \ldots, t_k\})$;
    let $P$ be a pushdown store; let $q_{act}$ mark the current state;
    create an alphabet symbol $a_{np}$ such that $a_{np} \notin \mathcal{A}$;
    create a pushdown store symbol $Z_{np}$ such that $Z_{np} \notin G$;
    create a state $q_{np}$ such that $q_{np} \notin Q$;
    let the pushdown store $P := (\#)$; let $i := 0$;
    if $\delta(q_0, |, \varepsilon) \neq (q_1, \varepsilon)$ then
        return error - no trees indexed in the input TCA;
    end
    while $P \neq ()$ do
        while $i < |pref\_bar(t)|$ and $pref\_bar(t)[i] \neq |$ do
            $a := pref\_bar(t)[i]$;
            if $a \in \mathcal{A}$ then push symbol $a$ on top of pushdown store $P$;
            else push symbol $a_{np}$ on top of pushdown store $P$;
            $i := i + 1$;
        end
        $i := i + 1$;
        $q_{act} := q_1$ ;
        repeat
            pop a symbol $b$ from the top of pushdown store $P$;
            if $b \notin \mathcal{A}$ and $b \neq a_{np}$ then
                if $\delta(q_{act}, \varepsilon, b) = (q_b, \varepsilon)$ then $q_{act} := q_b$;
                else $q_{act} := q_{np}$;
            end
        until $b \in \mathcal{A}$ or $b = a_{np}$;
        if $\delta(q_{act}, \varepsilon, b) = (q_b, c)$ then
            push symbol $c$ on top of the pushdown store $P$;
            report a match $(id_j, i)$ if $c = id_j$ for some $j$, $0 < j \leq k$
        else
            push symbol $Z_{np}$ on top of the pushdown store $P$;
        end
        if pushd. store $P$ contains only $s\#$, $s \notin \mathcal{A}$ and $s \neq a_{np}$ then set $P := ()$;
    end
end
```

the matched tree in the input. Thus the reported position of the matched subtree is the position of the last symbol of the matched subtree in the input.

To identify a subtree that is already indexed in a TCA, it is not necessary to add any state, pushdown store symbol or alphabet symbol to the TCA. That is why Algorithm 4.30 can avoid modifying TCA M and still report all subtrees that match any of the trees in $\{t_1, t_2, \ldots, t_k\}$. At the same time, if a state, a pushdown store symbol or an alphabet symbol has to be added to a TCA while a subtree is being read from input, then this tree was not yet indexed by the TCA and it cannot be one of the trees in $\{t_1, t_2, \ldots, t_k\}$. The subtree identifier of such subtree is thus not relevant. That is why Algorithm 4.30 does not need to modify TCA M. $\square$

Lemma 4.33 Let T be a set of trees. Let $id_1, id_2, \ldots, id_k$ be the subtree identifiers of trees in set T. Let t be a tree with n nodes. Assume a hash map is used for storing the transition function δ of $TCA(T)$. Assume a hash map is used for storing the subtree identifiers. Given $TCA(T)$, $id_1, id_2, \ldots, id_k$ and t for input, Algorithm 4.30 (TCA-matching) runs in $\mathcal{O}(2n)$ time.

Proof. The proof follows from the proof of Lemma 4.28, because Algorithm 4.30 is a modified Algorithm 4.12 (TCA-construction). The only change that can increase the complexity of the algorithm is at Step 27, where the subtree identifiers need to be searched for a match with a subtree identifier c. When a hash map is used, this operation runs in constant time. $\square$

The time required for finding all matching subtrees is an optimal result comparable with [32].

4.7 Exact repeats by TCA

The tree compression automaton can be easily used for searching subtree repeats in tree t. For every subtree t_s of t, a list of its occurrences in tree t can be created using an extension of Algorithm 4.12 (TCA-construction).

The algorithm works by simulating the $TCA(\{t\})$ automaton. Whenever the automaton reads a non-bar symbol from the input string, the index of the symbol is remembered. This symbol is a root node of some subtree of tree t. After the last symbol (the closing bar) of this subtree is read from the input string, the subtree is identified and the position of its root node is associated with the subtree identifier of the subtree.

This algorithm can be modified to accept $TCA(T)$ on the input, where t is a subtree of any of the trees in T. The size of its output is affected by the ratio $|TCA(T)|/|TCA(\{t\})|$. Its running time is not affected by this ratio if transition function δ is implemented by a lookup table.

Algorithm 4.34: Find all repeats in a tree

```
Name: TCA-repeats-search
Input: A tree $t$ and $TCA(\{t\}) = (Q, \mathcal{A}, G, \delta, q_0, \#, \varepsilon)$
Output: A relation $occ \subset S \times \mathbb{N}$, $S = G \setminus (\mathcal{A} \cup \{\#\})$. $(s, i) \in occ$ only if $s$ is a
        subtree identifier of a subtree $t_s$ of tree $t$ and $t_s$ has a root at index $i$ in
        $pref\_bar(t)$.
1  begin
2      let $P$ be a pushdown store; let $i$ be a counter;
3      set $i := 0$;
4      simulate automaton $TCA(\{t\})$ for input string $pref\_bar(t)$:
5      begin
6          whenever a transition $(q_0, a\alpha, \beta) \vdash (q_0, \alpha, a\beta)$ is made, increment $i$; if $a \neq |$,
             push the value of counter $i$ on top of pushdown store $P$;
7          whenever a transition $(q, \alpha, a\beta) \vdash (q_0, \alpha, j\beta)$ is made, pop a number $x$ from
             the top of the pushdown store and set $x \in occ(j)$;
8          when automaton $TCA(\{t\})$ accepts the input string, output $occ$ and exit;
9      end
10 end
```

Theorem 4.35 Relation *occ* defined by Algorithm 4.34 maps the position of every subtree root to a subtree identifier. If two subtrees are the same, the indices of their roots are mapped to the same subtree identifier.

Proof. Algorithm 4.34 (TCA-repeats-search) can be viewed as a modified Algorithm 4.12 (TCA-construction) that:

- pushes a pair $(a, index(a))$ on top of the pushdown store whenever a symbol $a \in \mathcal{A}, a \neq |$ is read from the input string
- pops a pair $(a, index(a))$ from the top of the pushdown store and replaces it there with a whenever the transition $(q, \alpha, a\beta) \vdash (q_0, \alpha, b\beta)$ is to be taken.

Since symbol b is the subtree identifier of the tree whose root is symbol a, $index(a) \in occ(b)$ is set for all roots a that are read from the input. This means that every subtree root is mapped to a subtree identifier.

Let t_s be a subtree of tree t that is the input of Algorithm 4.34. Let i_s be the subtree identifier of t_s. Let $TCA(\{t\})$ be in the configuration $(q_0, pref\_bar(t_s)\alpha, \beta)$. There is a sequence of transitions that $TCA(\{t\})$ can take ending in the configuration $(q_0, \alpha, i_s\beta)$. It must hold that $index(r_s) \in occ(i_s)$. If two subtrees are identical, their prefix bar notations are identical and therefore their subtree identifiers are identical. Therefore their root indices must be mapped to the same subtree identifier. □

4.7.1 Time and space complexity of Algorithm TCA-repeats-search

The time complexity depends directly on the complexity of Algorithm 4.12 (TCA-construction). It also depends on the complexity of the simulation of the TCA automaton and on the complexity of adding an element into $occ(b)$.

The TCA automaton is constructed in linear time if a lookup table or a hash map is used for transition function δ. Otherwise it is constructed in $n * log_2 n$ time. If n is the length of an input tree, the TCA automaton can be simulated in time either equal to n if a lookup table exists for the transition function δ, or $n * log_2 n$ otherwise.

The complexity of adding an element into $occ(b)$ is constant if a linked list is used for holding elements of $occ(b)$.

In total, the complexity of Algorithm 4.34 (TCA-repeats-search) follows the complexity of Algorithm 4.12 (TCA-construction): it is linear or $\mathcal{O}(n * log_2 n)$, depending on implementation of a lookup table by Algorithm 4.12.

The size of the output is $n * log_2 n$, which is the space required for storing pointers to the subtrees of tree t.

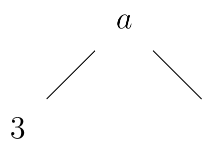
The total space required by Algorithm 4.34 again depends on the space required by the TCA automaton - the number of repeats is bounded by n.

4.8 Comparison with related compression methods

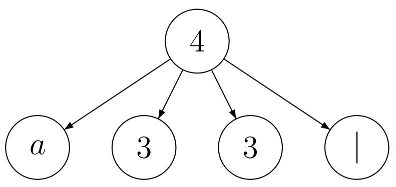
As stated in the Introduction, a similar approach to tree compression was investigated in [11]. There the result of tree compression is a grammar. The relationship between the two methods is shown on the example tree from Example 4.1. The grammar $G = (N = \{1, 2, 3, 4\}, T = \{a\}, P, 4)$ created by [11] that generates this tree has the following rules in P:

$4 \rightarrow a33|$
$3 \rightarrow a12|$
$2 \rightarrow a1|$
$1 \rightarrow a|$

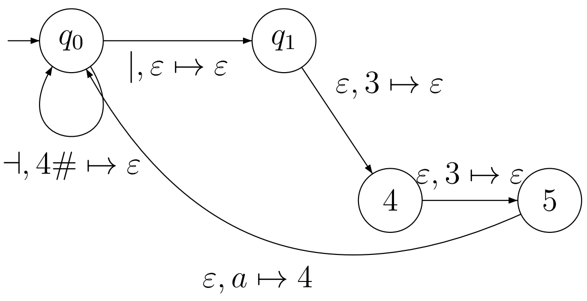
The pushdown store symbols of the TCA created for the example tree are the non-terminals of grammar G together with initial symbol $\#$. The right-hand sides of the rules of grammar G are preserved in the form of states and transition function δ. For example, when considering tree stub $(a, (3, 3))$, the rule $4 \rightarrow a33|$ corresponds to the states $q_1, 4, 5$ and transition function δ that involves them: $\delta(q_0, |) = q_1$, $\delta(q_1, 3) = 4$, $\delta(4, 3) = 5$, $delta(5, a) = q_0$. It holds that all words in the set $\{w : X \in N, X \Rightarrow^* w, w \in T^*\}$ are accepted by the TCA. If the smallest grammar extension from [11] is omitted, this set are exactly the words accepted by the TCA. Figure 4.9 illustrates this relationship.



Tree stub $(a, (3, 3))$



Grammar rule $4 \to a33|$


q_0
q_1
$|, \varepsilon \mapsto \varepsilon$
$\varepsilon, 3 \mapsto \varepsilon$
$\dashv, 4\# \mapsto \varepsilon$
4
$\varepsilon, 3 \mapsto \varepsilon$
5
$\varepsilon, a \mapsto 4$

Excerpt from the TCA(t_1) corresponding to the tree stub $(a, (3, 3))$

Figure 4.9: A tree stub, its rule in a straight line grammar, its states and transition rules in TCA

4.9 Implementation

The algorithms on tree compression automaton have been implemented as a C++ library [52]. The library can build a TCA from the provided tree in prefix bar notation. It can use the TCA as an index to find occurrences of the provided trees. It can decompress a TCA into the original tree. The library implements multiple options for representation of the transition function δ, including a hash map. The implementation verifies that using a hash map for representation of the transition function δ, the construction of TCA and its decompression takes time $\mathcal{O}(n)$ [52], where n is the number of nodes of the indexed tree. An interesting point is that in the performed measurements, decompression was approximately 4 times faster than compression. This difference is caused by the construction of the hash map during compression phase.

4.10 Experimental compression results

During experiments, the TCA was compressed into two separate sections: the transition table and a label lookup table. The transition table first indicates the incoming edges to each state that is not $q0$ or $q1$. Then it indicates all incoming edges to state $q0$. The label lookup table stores a mapping between node labels and internal alphabet symbols. The compressed binary file also contains a header that indicates how many bits a fixed-length

	Total	Stripped	BPLEX c.r.	mod. BPLEX c.r.	TCA c.r.
Swiss-Prot	77.27M	61.05M	2.43	3.72	2.55
Ligand Expo	16.95M	12.55M	4.58	5.54	4.49
xMark	11.88M	3.71M	3.48	7.61	6.25
Alpino Treeb.	2.54M	2.06M	4.27	5.58	3.00

Table 4.1: Compression performance compared to BPLEX; *total* = total size of XML files to compress; *stripped* = total size of XML files to compress, with text data removed from nodes

binary representation of an automaton state, alphabet symbol and subtree identifier takes. The compression achieved by TCA was compared with compression of a grammar-based algorithm BPLEX [11]. As these two approaches are close in nature, the compression ratios follow a similar curve. The compression performance was tested on pruned XML files, where text was removed from nodes. The sample data was retrieved from protein databases, linguistic records and was generated using specialised tools. The protein XML data was obtained from Swiss-Prot at UniProtKB [66] (first 1000 files) and Ligand Expo at Protein Data Bank [4] (also first 1000 files). Sample auction web-site XML data was generated by xMark [56], scaling factor was set to /f 0.1. Linguistic structures stored in XML were retrieved from the Alpino Treebank [61] (first 1000 files again). All sample XML files were stripped of text data (but not node labels) so that the tests measure solely the compression performance on the tree structures themselves.

The column BPLEX c.r. stands for compression ratio achieved by BPLEX, whereas modified BPLEX c.r. stands for compression ratio achieved by modified algorithm BPLEX which performs no further compression of the straight-line grammar.

The compression performance of the TCA is comparable with that of BPLEX; a worse performance is some cases is expected due to the fact that Algorithm 4.12 (TCA-construction) does not perform, compared to BPLEX, further compression of the generated straight-line grammar (especially notable on the xMark benchmark). However, the TCA outperforms BPLEX when compressing small XML files found in the Alpino Treebank database, possibly due to a better representation of the output. The same reason explains why the TCA outperforms BPLEX when compressing protein data from the Ligand Expo databank.

Algorithm 4.12 (TCA-construction) does not apply any compression on the produced TCA, similarly as the modified BPLEX algorithm does not to perform any further compression of the generated straight-line grammar. TCA outperforms the modified BPLEX. Due to similarities in nature between the TCA and BPLEX compression approaches, it is probable that after applying further compression to the generated automaton, TCA could perform as well or better than unmodified BPLEX.

4.11 Corresponding Finite Tree Automaton

Tree compression automaton can be converted to a deterministic finite tree automaton (DFTA) that accepts the same set of trees as the TCA. This section presents an algorithm that takes a TCA as input and constructs a DFTA which accepts the same set of trees as the TCA. It then proves that the constructed DFTA is deterministic and accepts the same set of trees.

Definition 4.36 Let T be a set of trees. Let $N = (Q_N, \mathcal{A}, Q_{Nf}, \delta_N)$ be an NFTA (see Definition 2.3). Let $M = (Q, \mathcal{A}_r, G, \delta, q_0, \#, \emptyset)$ be a $TCA(T)$. NFTA N and TCA M are equivalent if the set of all subtrees of trees in T is the set of trees accepted by NFTA N.

Algorithm 4.37: Transformation of TCA to equivalent DFTA

```
   Name: TCA-to-DFTA
   Input: $TCA(T) = (Q, \mathcal{A}, G, \delta, q_0, \#, \varepsilon)$
   Output: DFTA $M = (Q_M, \mathcal{A}_M, Q_{MF}, \delta_M)$ equivalent to $TCA(T)$
 1 begin
 2     construct grammar $R = (N, \mathcal{A} \cup \{|\}, P, S)$ from $TCA(T)$ using Algorithm 4.25
        (TCA-decompression);
 3     set $Q_M := \{\}$, $\delta_M := \{\}$, $\mathcal{A}_M = \{\}$;
 4     foreach rule $r = C \to a\ x_1 \dots x_n\ |, r \in P$, $a \in \mathcal{A}, x_1 \dots x_n \in G$ do
 5         set $Q_M := Q_M \cup \{C, x_1, \dots, x_n\}$;
 6         set $\mathcal{A}_M = \mathcal{A}_M \cup \{a_n\}$;
 7         set $\delta_M := \delta_M \cup \{a(x_1(y_1) \dots x_n(y_n)) \to C(a(y_1 \dots y_n))\}$;
 8     end
 9     set $Q_{MF} := Q_M$;
10 end
```

Example 4.38 Consider tree t_1 from Example 4.1. The $TCA(\{t_1\})$ constructed for this tree is shown in Figure 4.6. The DFTA equivalent with $TCA(\{t_1\})$, constructed by Algorithm 4.37, is a finite tree automaton $M = (\{q_1, q_2, q_3, q_4\}, \{a_2, a_1, a_0\}, \{q_1, q_2, q_3, q_4\}, \delta_M)$, where

$$\begin{aligned} \delta_M = \{\ a(3(y_1)3(y_2)) &\to 4(a(y_1y_2)), \\ a(1(y_1)2(y_2)) &\to 3(a(y_1y_2)), \\ a(1(y_1)) &\to 2(a(y_1)), \\ a() &\to 1(a())\ \}. \end{aligned}$$

Theorem 4.39 The NFTA M constructed by Algorithm 4.37 (TCA-to-DFTA) is deterministic (it is a DFTA).

Proof. Automaton M is not deterministic if there exists two different rules r_1, r_2 with the same left-hand side. This is only possible if $TCA(T)$ contains two different sequences of states $S_1 = (q_1, q_{11}, \dots, q_{1n})$ and $S_2 = (q_1, q_{21} \dots, q_{2m})$ such that $m = n$, $(q_1, \varepsilon, a) \to (q_{11}, \varepsilon) \in \delta$, $(q_{1i}, \varepsilon, Z_i) \to (q_{1(i+1)}, \varepsilon) \in \delta$ and $(q_1, \varepsilon, a) \to (q_{21}, \varepsilon) \in \delta$, $(q_{2i}, \varepsilon, Z_i) \to (q_{2(i+1)}, \varepsilon) \in \delta$ for all $0 < i \leq m$. If the sequences are indeed different, this implies that there exists a state q in $TCA(T)$ such that $\{(q, \varepsilon, Z) \to (q', \varepsilon), (q, \varepsilon, Z) \to (q'', \varepsilon)\} \in \delta$, $q' \neq q''$. But that would mean that $TCA(T)$ is non-deterministic, which is not true. Thus NFTA M is deterministic. □

In other words, automaton M would be non-deterministic if there was a subtree identifier in $TCA(T)$ that marked two different trees. But that is not possible (see Theorem 4.6).

Theorem 4.40 Given a set of trees T, automaton $TCA(T)$ and DFTA M constructed by Algorithm 4.37 (TCA-to-DFTA) with input $TCA(T)$ are equivalent.

Proof. It has to be shown that DFTA M accepts *all* trees in T and all their subtrees and *no other* tree.

ALL: Grammar R constructed from $TCA(T)$ has been proved to generate all subtrees of trees in T in prefix bar notation. If rule $r = C \to a|$, then new rule $a \to C(a)$ is added to δ_M. Since state C is final, tree $a|$ is accepted by DFTA M. By induction, if rule $r = C \to ax_1 \dots x_n|$, then subtrees generated by grammar R from non-terminal symbols $x_1 \dots x_n$ are reduced by DFTA M into the corresponding states $x_1, \dots, x_n$. Since rule $a(x_1(y_1) \dots x_n(y_n)) \to C(a(y_1 \dots y_n))$ exists in DFTA M, tree with subtree identifier C is accepted by DFTA M. Thus all subtrees of trees in T are accepted by M.

NO OTHER: Assume that tree t is the smallest tree that is accepted by DFTA M but not by $TCA(T)$. If $pref\_bar(t) = a|$ for some $a \in \mathcal{A}$, then rule $C \to a$ for some $C \in G$ must exist in grammar R. But then tree $a|$ is accepted by $TCA(T)$, which is a contradiction.

If $pref\_bar(t) = a\ pref\_bar(t_1) \dots pref\_bar(t_n)\ |$, then trees $t_1, \dots t_n$ are accepted by $TCA(T)$. These trees reduce to states $q_1, \dots q_n$ by application of transition function δ_M of DFTA M. If M accepts tree t by state q_f, then there must exist a rule $r = a(q_1(y_1) \dots q_n(y_n)) \to q_f(a(y_1 \dots y_n))$, $r \in \delta_M$. Rule r is a member of δ_M only if at Step 6 rule r is added to δ_M, based on rule $r_{orig} = q_f \to a\ q_1 \dots q_n\ |$ of grammar R. But if r_{orig} is a rule of grammar R, then $TCA(T)$ must accept tree with subtree identifier q_f. This is a contradiction. □

The existence of Algorithm 4.37 for conversion between TCA and NFTA implies that finite tree automaton is suitable for subtree indexing, for compression of trees and for finding exact repeats of subtrees. The latter two applications have not been explored in [18] and TCA is thus a contribution to the theory presented there.

Chapter 5

A Full and Linear Index of a Tree for Tree Patterns

Tree pattern searching with an index assumes a single subject tree t, of which an index is built to improve performance of repeated searching of patterns. A tree pattern p is input; this pattern is searched in the subject tree t using the index. Tree pattern p is a tree with some of its leaves replaced by a special symbol S, which is a placeholder for any subtree. The goal of tree pattern searching is to find all subtrees of tree t that match tree pattern p [3]. When using linear tree notations, tree pattern searching can be treated as a variant of string searching with variable length gaps [54]. Using linearised trees, symbols S represent gaps in the linearised tree pattern p. The gaps are of unknown size and a condition is placed on the matched string to be a linear representation of some tree.

This chapter presents a solution for building a tree index for ranked tree patterns and for searching ranked tree patterns from paper [A.2]. Given a subject ranked tree t with n nodes, the tree is preprocessed and an index, which consists of a standard text compact suffix automaton and a subtree jump table, is constructed. The number of distinct tree patterns which match the tree is $\mathcal{O}(2^n)$, but the size of the index is $\mathcal{O}(n)$. The searching phase reads an input tree pattern p of size m and locates all its occurrences in tree t. For an input tree pattern p in linear prefix notation $pref(p) = p_1Sp_2S \dots Sp_k$, $k \geq 1$, the searching is performed in time $\mathcal{O}(m + \sum\limits_{i=1}^{k} |occ(p_i)|)$, where $occ(p_i)$ is the set of all occurrences of p_i in $pref(t)$.

The chapter consists of four sections. The first section deals with the preprocessing phase, in which an index of a subject tree is constructed. The second section describes the searching phase, in which tree pattern occurrences of an input tree pattern are located using the index. The third section describes the time and space complexities of both phases. The last section shows that the presented algorithms for construction and usage of the index can be viewed as a simulation of a non-deterministic tree pattern PDA.

5.1 Construction of Index

The index of a tree t consists of two parts:

– A compact suffix automaton [20] for $pref(t)$, which accepts all substrings of $pref(t)$. We note that there are substrings of $pref(t)$ which are not subtrees of t in the prefix notation.

– A *subtree jump table*, a linear-size structure needed for finding ends of subtrees represented by special symbols S.

The preprocessing phase constructs the two parts of the index. It also constructs and initializes a working data structure (called *Pairs*) that is used during the searching phase.

Definition 5.1 Let t and $pref(t) = a_1a_2 \ldots a_n$, $n \geq 1$, be a tree and its prefix notation, respectively. A *subtree jump table* $SJT(t)$ is defined as a mapping from set $\{1..n\}$ into set $\{2..n+1\}$. If $a_ia_{i+1} \ldots a_{j-1}$ is the prefix notation of a subtree of tree t, then $SJT(t)[i] = j$, $1 \leq i < j \leq n+1$.

Algorithm 5.2: Construction of subtree jump table

```
Name: SJT-construction
Input: Tree $t$ in prefix notation $pref(t)$, index of current node $rootIndex$ (default
       value = 1), reference to an (initially empty) subtree jump table $SJT(t)$
Output: index $exitIndex$, subtree jump table $SJT(t)$
1 begin
2     $index = rootIndex + 1$;
3     for $i = 1$ to $Arity(pref(t)[rootIndex])$ do
4         $index$ = SJT-construction($pref(t), index, SJT(t)$);
5     end
6     $SJT(t)[rootIndex] = index$;
7     return $index$;
8 end
```

Lemma 5.3 Given $pref(t)$ and $rootIndex$ equal to 1, Algorithm 5.2 constructs subtree jump table $SJT(t)$.

Proof. The algorithm terminates because every recursive step is for the next symbol from $pref(t)$, starting with the first symbol.

Property of SJT: Each pair $(first, last)$ of the subtree jump table maps a symbol at index $first$ to index $last$. If m is the number of nodes of subtree rooted at node at index $first$, then $last = first + m$.

Assume Algorithm 5.2 is invoked with input $pref(t)$ and $rootIndex = 1$. Then, at every entry to line 2 of the algorithm the index $first = rootIndex$ is equal to index of the root of the currently processed subtree because $rootIndex$ is incremented for every symbol of $pref(t)$ in the left-to-right order.

For subtrees that consist of a single node, Property of SJT is satisfied if the algorithm is called with the correct index $first$ of the root node in the prefix notation. After return, $exitIndex - rootIndex = 1$ is equal to the size of the processed tree.

For subtrees of greater size than 1, the Property of SJT is satisfied because the index $exitIndex = last$ is equal to $first$ plus one (for the root node) plus the sum of the sizes of the child subtrees of node at index $rootIndex$. $\square$

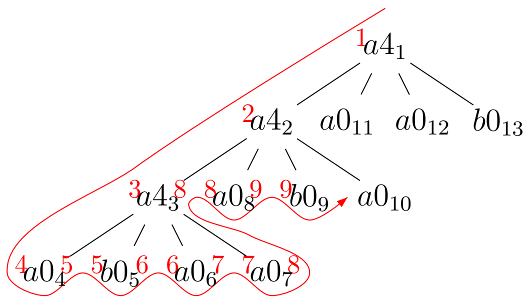


Figure 5.1: Subtree jump table construction for tree t_1 from Example 5.4

Example 5.4 Consider tree t_1 over $\mathcal{A}$ from Example 2.1, $pref(t_1) = a4_1a4_2a4_3a0_4$ $b0_5a0_6a0_7$ $a0_8b0_9a0_{10}$ $a0_{11}a0_{12}b0_{13}$.

– Compact suffix automaton $Mc(pref(t_1))$ [20] is illustrated in Figure 5.2.

– Subtree jump table $SJT(t_1)$, constructed by Algorithm 5.2, is in Table 5.1. The idea behind construction of Subtree jump table is illustrated in Figure 5.1.

source	1	2	3	4	5	6	7	8	9	10	11	12	13
target	14	11	8	5	6	7	8	9	10	11	12	13	14

Table 5.1: Subtree jump table for tree t_1 from Example 5.4

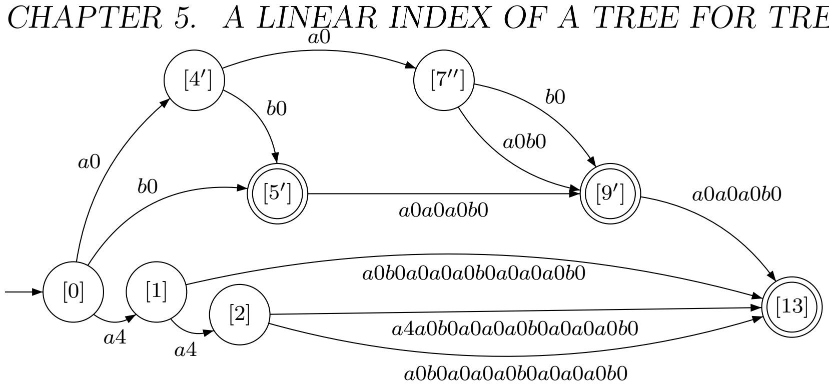


Figure 5.2: Transition diagram of compact suffix automaton $Mc(pref(t_1))$ for tree t_1 from Example 5.4; the long edge labels can be replaced by pairs of beginning and ending indexes into $pref(t_1)$

Furthermore, the array $Pair^n_{\mathcal{P}}$ serves as a working data structure for the main searching algorithm during the searching phase and its initial value, denoted $Pair^n_{\{\}}$, is to be set once.

Definition 5.5 Let $\mathcal{P} = \{(first_1, last_1), \dots (first_k, last_k)\}$ be a set of pairs of positive integers such that $last_i \neq last_j$ if $i \neq j$, $1 \leq i \leq k$, $1 \leq j \leq k$. Array $Pair^n_{\mathcal{P}}$ is an array of integers such that $Pair^n_{\mathcal{P}}[last_h] = first_h$ for all $1 \leq h \leq k$. For all other values $1 \leq v \leq n$, $Pair^n_{\mathcal{P}}[v] = -1$.

The array $Pair^n_{\mathcal{P}}$ will be used for representing occurences of subtrees in the prefix notation of a subject tree. The array is structured in a way that allows fast lookups of the index $first$ given an index $last$.

Example 5.6 Array $Pair^{13}_{\{(1,11),(2,8),(3,5)\}}$, which represents occurrences of prefix $a4\ S$ of tree pattern p'' from Example 2.2 in tree t_1 from Example 2.1, is illustrated in Table 5.2.

index	1	2	3	4	5	6	7	8	9	10	11	12	13
value	-1	-1	-1	-1	3	-1	-1	2	-1	-1	1	-1	-1

Table 5.2: Array $Pair^{13}_{\{(1,11),(2,8),(3,5)\}}$ from Example 5.6

5.2 Searching occurrences of input tree patterns

Algorithm 5.23 (SearchPattern - see below) finds all occurrences of an input pattern p in tree t. The algorithm uses three sub–algorithms that are presented before the searching algorithm: Algorithm 5.18 (VerifyArityChecksum), Algorithm 5.19 (FindOccurrences) and Algorithm 5.21 (MergeOccurrences).

Definition 5.7 Let $pref(p) = p_1 S p_2 S \dots S p_k$ be the prefix notation of a tree pattern p over an alphabet $A \cup \{S\}$, where no substring p_i, $1 \leq i \leq k$, contains any symbol S. The substring p_i is called a *sub-pattern* of p at index i.

Example 5.8 Consider $pref(p'') = a4Sa0SS$, the prefix notation of tree pattern p'' from Example 2.2. Tree pattern p'' has four sub-patterns, $pref(p'') = p_1Sp_2Sp_3Sp_4$, where $p_1 = a4$, $p_2 = a0$, $p_3 = \varepsilon$ and $p_4 = \varepsilon$.

Definition 5.9 Let $pref(t) = a_1a_2...a_n$ be the prefix notation of a tree t. Let $pref(p) = p_1Sp_2S...p_k$ be the prefix notation of a tree pattern p. An *occurrence* of sub-pattern p_i in $pref(t)$ is a pair $(first, last)$, where:

- if $p_i = \varepsilon$, $1 < first = last \leq n + 1$,
- if $p_i \neq \varepsilon$, $1 \leq first < last \leq n + 1$ and $a_{first}a_{first+1} \dots a_{last-1} = p_i$.

The set of all occurrences of sub-pattern p_i in $pref(t)$ is denoted by $occ^t(p_i)$. If tree t is obvious from the context, the set can be denoted by $occ(p_i)$.

Example 5.10 Consider sub-pattern $p_2 = a0$ of tree pattern p'' from Example 2.2. Sub-pattern p_2 has seven occurrences in tree t: $occ^t(p_2) = \{\ (4,5), (6,7), (7,8), (8,9), (10,11), (11,12), (12,13)\ \}$.

Definition 5.11 Let $pref(p) = p_1Sp_2S \dots Sp_k$ be the prefix notation of a tree pattern p. Then any string $p_1Sp_2S \dots Sp'_k$, $k' \leq k$, or $p_1Sp_2S \dots Sp''_kS$, $k'' < k$, is called a *tree pattern prefix* of tree pattern p, abbreviated $TPP(p)$.

Example 5.12 Consider tree pattern p'', $pref(p'') = a4Sa0SS$, from Example 2.2. Then $\{a4, a4S, a4Sa0, a4Sa0S, a4Sa0SS\}$ is a set of tree pattern prefixes of tree pattern p''.

Definition 5.13 Let p be a tree pattern and t be a tree. An *occurrence of tree pattern prefix* $TPP(p) = p_1Sp_2S \dots Sp_k$ in tree t is a pair $(first, last)$, where $(first, last_1)$ is an occurrence of sub-pattern p_1 in $pref(t)$, pair $(SJT(t)[last_1], last_2)$ is an occurrence of sub-pattern p_2 in $pref(t)$, $\dots$, and pair $(SJT(t)[last_{k-1}], last)$ is an occurrence of sub-pattern p_k in $pref(t)$. The set of all occurrences of a tree pattern prefix $TPP(p)$ in $pref(t)$ is denoted by $occ^t(TPP(p))$. If tree t is obvious from the context, the set can be denoted by $occ(TPP(p))$.

Example 5.14 Consider tree pattern prefix $TPP_1(p'') = a4S$ of tree pattern p'' from Example 2.2. Consider tree t_1 from Example 2.1. Then $occ^{t_1}(TPP_1(p'')) = \{(1, 11), (2, 8), (3, 5)\}$.

Lemma 5.15 Let $(first, last)$ be an occurrence of a tree pattern prefix $pref(p)$ in a tree t, $pref(p) = p_1Sp_2 \dots Sp_k$, $pref(t) = a_1a_2 \dots a_{first}a_{first+1} \dots\ a_{last-1}a_{last} \dots a_n$. Then pattern p matches tree t at node a_{first}. Node a_{last-1} is the rightmost leaf of the subtree rooted at node a_{first}.

Proof. Prefix notation of subtree rooted at node a_{first} is $a_{first} \dots a_{first+m-1}$. By definition of occurrence of sub-pattern, $a_{first}a_{first+1} \dots a_{first+|p_1|-1} = p_1$. Pair $(a_{first}, a_{first+|p_1|})$ is the occurrence of tree pattern prefix p_1. If $pref(p) = p_1$, then tree pattern p matches tree t at node a_{first} and it holds that $a_{first+|p_1|-1} = a_{last-1} = a_{first+m-1}$ is the rightmost leaf of subtree p.

The proof proceeds by induction.

Let (a_{first}, a_{last_i}) be an occurrence of tree pattern prefix $p_1S \dots Sp_i$, $1 \leq i < k$. By definition of subtree jump table, it holds that $a_{last_i} \dots a_{last_i+j-1}$ is the prefix notation of a subtree t_i of j nodes, rooted at a_{last_i}. By definition of occurrence of tree pattern prefix, $a_{last_i+j} \dots a_{last_i+j+|p_{i+1}|-1} = p_{i+1}$. Pair $(a_{first}, a_{last_i+j+|p_{i+1}|})$ is then an occurrence of tree pattern prefix $p_1S \dots p_iSp_{i+1}$.

If $i+1 = k$, then $a_{first}a_{first+1} \dots a_{last_i+j} \dots a_{last_i+j+|p_{i+1}|-1}$ is the prefix notation of a tree obtained from tree pattern p by substituting a subtree $t_{i'}$ for the i'-th occurrence of symbol S in p, $i' = 1, 2, \dots, i$. Thus tree pattern p matches tree t at node a_{first}. The rightmost leaf of the matched subtree is $a_{last_i+j+|p_{i+1}|-1} = a_{last-1} = a_{first+m-1}$. □

We note that an *occurrence of tree pattern* p in tree t is an occurrence of tree pattern prefix $pref(p)$ in $pref(t)$.

Example 5.16 Consider tree pattern p'' from Example 2.2. Tree pattern p'' has two occurrences in tree t_1: $occ^{t_1}(p'') = \{(1,14),(2,11)\}$.

Lemma 5.17 Let t be a tree and p be a tree pattern. Let pairs $(first_A, last_A)$ and $(first_B, last_B)$, $first_A \neq first_B$, be occurrences of tree pattern prefix $TPP(p) = p_1Sp_2S\dots$ in tree t. If $TPP(p) \neq pref(p)$, then $last_A \neq last_B$.

Proof. If $TPP(p) = p_1$ and $TPP(p) \neq pref(p)$, then $last_A = first_A + |p_1|$ and $last_B = first_B + |p_1|$. Then it is true that $last_A \neq last_B$.

The proof proceeds by mutual induction.

- If $TPP(p) = p_1Sp_2S \dots p_kS$, then there exist occurrences $(first_A, last'_A)$ and $(first_B, last'_B)$, $last'_A \neq last'_B$, of tree pattern prefix $TPP'(p) = p_1Sp_2\ S \dots p_k$. There must exist two subtrees t_A, t_B of tree t, $pref(t_A) = a_{last'_A}\ a_{last'_A+1} \dots a_{last'_A+i}$ and $pref(t_B) = a_{last'_B}a_{last'_B+1} \dots a_{last'_B+j}$, $last'_A + i = last_A - 1$, $last'_B + j = last_B - 1$. If $last'_A + i = last'_B + j$, then t_B is a subtree of t_A or vice versa.

  Let us assume, without loss of generality, that tree t_B is a subtree of tree t_A. Then no node on the path *path* from $root(t_A)$ to $root(t_B)$ has a right sibling (because $last'_A + i = last'_B + j$). Since $root(t_A) \neq root(t_B)$, *path* is of non-zero length. By induction, this implies that any node on the path from the node at position $last'_A + i - 1 = last_A - 1$ to the node at position $first_A$ has no right sibling. But this implies that $TPP(p) = pref(p)$, which contradicts the initial condition.

- If $TPP(p) = p_1S \dots p_{k-1}Sp_k$, then there exist occurrences $(first_A, last'_A)$ and $(first_B, last'_B)$, $last'_A \neq last'_B$, of tree pattern prefix $TPP'(p) = p_1Sp_2\ S \dots p_{k-1}S$. Since $last_A = last'_A + |p_k|$ and $last_B = last'_B + |p_k|$, it holds that $last_A \neq last_B$.

□

Algorithm 5.18: Verification of arity checksum from [50]

```
Name: VerifyArityChecksum
Input: String over a ranked alphabet $str = a_1 a_2 \dots a_n$, $n \geq 1$.
Output: Decision whether $str = pref(t)$ for some tree $t$.
begin
    Set $ac(str) := 1$;
    /* ac(str) stands for arity checksum of a string $str$    */
    for $i := 1$ to $n$ do
        $ac(str) := ac(str) + Arity(a_i) - 1$;
        if $i < n$ and $ac(str) = 0$ then
            return $false$;
    end
    if $ac(str) = 0$ then
        return $true$;
    return $false$;
end
```

Algorithm 5.19: Finding Occurrence of Sub-patterns

```
Name: FindOccurrences
Input: Compact suffix automaton $Mc(pref(t))$, $|pref(t)| = n$, sub-pattern $p_i$ of
       tree pattern $p$
Output: $occ^t(p_i)$
begin
    Let $q$ be the state of the $Mc$ reached after processing $p_i$ from input;
    Find all paths from state $q$ that lead to a final state of $Mc$;
    For each path of length $length$ from state $q$ to a final state, there is an
      occurrence $(n - length - |p_i|, n - length)$ of sub-pattern $p_i$;
end
```

Lemma 5.20 Given a string $str = a_1 a_2 \dots a_n$ on the input, Algorithm 5.18 (VerifyArityChecksum) decides whether $str = pref(t)$ for a tree t.

Proof. If $str = pref(t)$ for a tree t, then:

If $Arity(a_1) = 0$, arity checksum is 0 and the algorithm outputs true. For $pref(t) = a_1\ pref(t_1) \dots pref(t_k)$ for some trees $t_1 \dots t_k$ and $Arity(a_1) = k$, the algorithm outputs true for input $pref(t_1)$, true for input $pref(t_2)$, ... and true for input $pref(t_k)$. This means that arity checksum ac of any of these strings is 0 and it never is 0 for any prefix of these strings. Since it holds that $ac(pref(t)) = Arity(root(t)) + ac(pref(t_1)) - 1 + \cdots + ac(pref(t_k)) - 1$, the arity checksum $ac(pref(t)) = 0$ and the algorithm outputs true.

If $str \neq pref(t)$ for any tree t, then:

- If $|str| = 1$, then $k \neq 0$ and the algorithm outputs false.
- Otherwise consider substring sub of str, $sub = a_1a_2 \ldots a_n$. If there does not exist a prefix $psub$ of sub, $psub = a_1a_2 \ldots a_m$, $m \leq n$ such that $psub = pref(t_1)$ for some t_1, then by induction the algorithm outputs false. If such prefix $psub$ exists, set $sub = a_{m+1}a_{m+2} \ldots a_n$ and apply same reasoning to substring $a_m, a_{m+1} \ldots a_n$ until $m = n$. When $m = n$ and substring sub has been successfully divided into prefix notations of k' subtrees, then the algorithm outputs false if $k' \neq k$. If the algorithm outputs true, then $k' = k$ and $str = pref(t)$ for some tree t - but this contradicts the original assumption that $str \neq pref(t)$.

□

Algorithm 5.21: Merging Occurrences

```
Name: MergeOccurrences
Input: A set $prevOcc = occ^t(TPP(p))$, a set $subOcc = occ^t(p_k)$, an array
       $Pair^{|pref(t)|}_{\{\}}$
Output: A set $mergedOcc = occ^t(TPP(p)p_k)$
1 begin
2     $mergedOcc := \{\}$;
3     foreach $(first, last)$ in $prevOcc$ do $Pair^{|pref(t)|}_{prevOcc}[last] := first$ ;
4     foreach $(first', last')$ in $subOcc$ do
5         if $Pair^{|pref(t)|}_{prevOcc}[first'] \neq -1$ then
            $mergedOcc := mergedOcc \cup \{(Pair^{|pref(t)|}_{prevOcc}[first'], last')\}$ ;
6     end
7     foreach $(first, last)$ in $prevOcc$ do $Pair^{|pref(t)|}_{\{\}}[last] := -1$ ;
8     return $mergedOcc$;
9 end
```

Theorem 5.22 Let $TPP'(p) = p_1Sp_2S \ldots Sp_{k-1}Sp_k$ be a tree pattern prefix of a tree pattern p. Let $prevOcc = occ^t(TPP(p))$ be a set of occurrences of a tree pattern prefix

$TPP(p) = p_1Sp_2S\dots Sp_{k-1}S$; let $subOcc = occ^t(p_k)$ be a set of occurrences of a sub-pattern p_k. Given $prevOcc$ and $subOcc$ on input, Algorithm 5.21 (MergeOccurrences) computes occurrences $mergedOcc = occ^t(TPP'(p))$ of tree pattern prefix $TPP'(p)$.

Proof. The proof is by contradiction. Throughout the proof, set $Pair$ is used. Thanks to Lemma 5.17, indexes $last$ in set of pairs $prevOcc$ are known to be pairwise different and thus no conflicts in use of set $Pair$ can occur.

1. An occurrence $(first_{inv}, last_{inv})$ is output in set $mergedOcc$ although it is not an occurrence of tree pattern prefix $TPP'(p)$.

   The occurrence $(first_{inv}, last_{inv})$ must have been included in $mergedOcc$ on line 5 of the algorithm, $(first_{inv}, last_{inv}) = (Pair^{|pref(t)|}_{prevOcc}[first'], last')$ for some index $first'$. That means that an occurrence $(first', last')$ is an occurrence of sub-pattern p_k, which must be present in $occ^t(p_k)$.
   Since $Pair^{|pref(t)|}_{prevOcc}[first'] \neq -1$, an occurrence $(Pair^{|pref(t)|}_{prevOcc}[first'], first')$ is an occurrence of tree pattern prefix $TPP(p)$ and it must be present in $occ^t(TPP(p))$. Since $TPP'(p)$=$TPP(p)p_k$, it holds that $first_{inv} = Pair^{|pref(t)|}_{prevOcc}[first']$, $last_{inv} = last'$, $(first_{inv}, last_{inv}) \in occ^t(TPP'(p))$.

2. An occurrence $(first_{inv}, last_{inv})$ of tree pattern prefix $TPP'(p)$ is not output in the set $mergedOcc$.

   Since $TPP'(p) = TPP(p)p_k$, there must exist an index $last$ such that $(first_{inv}, last)$ is an occurrence of $TPP(p)$ and $(last, last_{inv})$ is an occurrence of p_k: $(first_{inv}, last) \in prevOcc$ and $(last, last_{inv}) \in subOcc$. Because of Lemma 5.17 and lines 3 and 5 of the algorithm, $(first_{inv}, last_{inv}) \in mergedOcc$.

□

A linear time merging algorithm would be simple if the sets of occurrences $(first, last)$ were in the form of lists sorted by index $first$ or by index $last$. Such a principle is used in related work [7]. Unfortunately, we do not know how to gain a sorted list of occurrences from the compact suffix automaton in a linear time and therefore we avoid sorting completely, reaching the linear time for merge operation by other means.

Algorithm 5.23 (SearchPattern) finds all occurrences of a pattern p in tree t. It calls these sub-algorithms:

– Algorithm 5.18 (VerifyArityChecksum) – this algorithm computes *arity checksum* of the provided tree pattern p to make sure that p is a valid tree pattern. The arity checksum of the whole pattern must be equal to 0, while the arity checksums of all non-empty proper prefixes of $pref(p)$ must be greater than 0 [50].

Algorithm 5.23: Searching for occurrences of a tree pattern

```
Name: SearchPattern
Input: Tree pattern $p$, $pref(p) = p_1 S p_2 S \dots p_k$, compact suffix automaton
       $Mc(pref(t))$, subtree jump table $SJT(t)$, Array $Pair_{\{\}}^{|pref(t)|}$
Output: List of occurrences of tree pattern $p$
begin
    if VerifyArityChecksum($p$) = false then
        return $ERROR$ – $invalid\ pattern$;
    end
    $prevOcc := \{\}$;
    for $i := 1$ to $k$ do
        if $p_i \neq \varepsilon$ then
            $occ$ := FindOccurrences($Mc$,$p_i$);
            if $i = 1$ then $prevOcc := occ$ ;
            else $prevOcc$ := MergeOccurrences($prevOcc$,$occ$,$Pair_{\{\}}^{|pref(t)|}$) ;
        end
        if $i \neq k$ then
            foreach occurrence $(first, last)$ in $prevOcc$ do
                $(first, last) := (first, SJT(t)[last])$;
            end
        end
    end
    return $prevOcc$;
end
```

– Algorithm 5.19 (FindOccurrences) – this is a substring searching algorithm from [20]. This algorithm is reproduced in Listing 5.19 with some technical details omitted for the sake of brevity and clarity.

– Algorithm 5.21 (MergeOccurrences) – this is an algorithm that merges two sets of occurrences in linear time.

Example 5.24 Consider the prefix notation $pref(p'') = a4Sa0SS$ of tree pattern p'', illustrated in Figure 2.2. Tree pattern p'' can be rewritten as $pref(p'') = p_1Sp_2Sp_3Sp_4$, where $p_1 = a4$, $p_2 = a0$ and $p_3 = p_4 = \varepsilon$.

Consider the run of Algorithm 5.23 (SearchPattern) using tree pattern p'', compact suffix automaton $Mc(pref(t_1))$ and subtree jump table SJT(t_1):

Algorithm 5.18 (VerifyArityChecksum) returns true for tree pattern p'' because p'' is a valid tree pattern (if you replaced symbols S with $a0$ symbols in the prefix notation of the pattern, you would get a prefix notation of a tree).

At $i = 1$, after Algorithm 5.19 (FindOccurrences) is executed, $prevOcc = \{(1, 2), (2, 3), (3, 4)\}$. Using subtree jump table $SJT(t_1)$, $prevOcc$ is then rewritten to $prevOcc = \{(1, 11), (2, 8), (3, 5)\}$.

At $i = 2$, after Algorithm 5.19 is executed, $occ = \{((4, 5), (6, 7), (7, 8), (8, 9), (10, 11), (11, 12), (12, 13)\}$. Using Algorithm 5.21 (MergeOccurrences), $prevOcc$ is rewritten to $prevOcc = \{(1, 12), (2, 9)\}$. Using $SJT(t_1)$, $prevOcc$ is then rewritten to $prevOcc = \{(1, 13), (2, 10)\}$.

At $i = 3$, algorithm uses $SJT(t_1)$ to rewrite $prevOcc$ to $prevOcc = \{(1, 14), (2, 11)\}$.

At $i = 4$, $prevOcc$ is not modified because sub-pattern p_4 is the empty string and the algorithm returns set of occurrences $\{(1, 14), (2, 11)\}$.

Algorithm 5.23 has found two occurrences of tree pattern p'': the first one starting at position 1 (ending at position 14) and the second one at position 2 (ending at position 11) in $pref(t_1)$.

Theorem 5.25 Algorithm 5.23 (SearchPattern) finds all occurrences $occ^t(p)$ of tree pattern $p = p_1Sp_2S \dots Sp_k$ in tree t.

Proof. First, the algorithm verifies validity of the provided tree pattern p prior to continuing. This checking simplifies reasoning about the rest of the algorithm.

The algorithm then proceeds as follows: it iteratively finds all occurrences of tree pattern prefix $p'_1 = p_1S$, then $p'_2 = p_1Sp_2S$ and so on until $p'_{k-1} = p_1Sp_2S \dots Sp_{k-1}S$. Eventually, it finds occurrences of tree pattern $p = p_1Sp_2S \dots Sp_k$ in tree t.

There are two cases to disprove:

1. An occurrence $(first, last)$ of pattern p is not output in the set of all occurrences.

   If $k = 1$, the algorithm outputs exactly the output of the compact suffix automaton, including $(first, last)$. This contradicts the assumption.

   For all $k > 1$, the algorithm is able to find all occurrences $occ^t(p'_{k-1})$ of tree pattern prefix $p'_{k-1} = p_1Sp_2S \dots Sp_{k-1}S$. This is achieved by finding all occurrences

$(first', last')$ of tree pattern prefix $p_1Sp_2S\dots Sp_{k-1}$ and then (line 13) by using subtree jump table to extend these occurrences to occurrences $(first', SJT(t)[last'])$ of tree pattern prefix p'_{k-1}. Using compact suffix automaton $Mc(pref(t))$, the algorithm finds all occurrences $occ^t(p_k)$ of sub-pattern p_k. The two sets of occurrences are then given to Algorithm 5.21 (MergeOccurrences), which was proved to output a set of occurrences $occ^t(p'_{k-1}p_k)$. Since $pref(p) = p'_{k-1}p_k$ and $(first, last)$ must be equal to some $(first', SJT(t)[last'] + |p_k|)$, the occurrence $(first, last)$ must be present in the returned set $prevOcc$, which contradicts the original assumption.

2. Algorithm outputs a pair $(first, last)$ in set $occ^t(p)$ that is not an occurrence of tree pattern p in tree t.

   If $k = 1$, this cannot happen, because a compact suffix automaton is used to generate the pair $(first, last)$.

   For all $k > 1$, the algorithm can find exactly the occurrences of tree pattern prefix $p_1Sp_2S\dots Sp_{k-1}$ and, using the subtree jump table, also exactly the occurrences $occ^t(p'_{k-1})$ of tree pattern prefix $p'_{k-1} = p_1Sp_2S\dots Sp_{k-1}S$. Using the compact suffix automaton $Mc(pref(t))$, the algorithm finds exactly the occurrences $occ^t(p_k)$ of sub-pattern p_k. Algorithm 5.21 (MergeOccurrences) was proved to output only occurrences of tree pattern p for these inputs p_{k-1} and p_k. Since at this point $i = k$, only occurrences of pattern p are output. This contradicts the original assumption.

□

5.3 Time and space complexities

Lemma 5.26 Algorithm 5.2 (SJT-construction) runs in $\mathcal{O}(n)$ time, where n is the number of nodes of the subject tree t.

Proof. The algorithm is based on a depth-first search traversal of the subject tree, where at each node only a constant amount work is performed (line 7). Thus, its running time is bound by the number of nodes n. Counting assignment operations, the running time is at worst $7n$. □

Theorem 5.27 Construction of index of a tree t (construction of subtree jump table and of compact suffix automaton) takes time $\mathcal{O}(n)$ and produces an index of size $\mathcal{O}(n)$.

Proof. The creation of compact suffix automaton of size $\mathcal{O}(n)$ [20] and the creation of an array of integers of size n require $\mathcal{O}(n)$ time. Algorithm 5.2 that creates the subtree jump table is proved to be linear in time and space in Lemma 5.26. The array $Pairs$ is created in time $\mathcal{O}(n)$. Following from the definitions, the size of array $Pairs$ is n and the size of the subtree jump table is also n. Thus the size of the whole index is $\mathcal{O}(n)$ and it is created in $\mathcal{O}(n)$ time. □

Lemma 5.28 Algorithm 5.21 (MergeOccurrences) runs in $\mathcal{O}(|prevOcc|+|occ|)$ time, where $|prevOcc| + |occ|$ is the number of occurrences in both input sets.

Proof. The algorithm uses an extra memory of size n prepared during the indexing phase. This memory allows for the fast lookup used by the algorithm. The algorithm runs in three loops whose lengths are determined by $|prevOcc| + |occ|$ and at each iteration in each loop, the amount of work is constant. Thus, the total running time holds. Counting assignment operations, the running time is at most $1 + 2|prevOcc| + min(|occ|, |prevOcc|)$. □

Theorem 5.29 Let $pref(p) = p_1 S p_2 S \dots S p_k$ of total length m be the prefix notation of a tree pattern p. Algorithm 5.23 (SearchPattern) runs in $\mathcal{O}(m + \sum_{i=1}^{k} |occ'(p_i)|))$ time, where:

- $occ'(p_i) = occ(p_i)$ if $p_i \neq \varepsilon$,
- $occ'(p_i) = occ'(p_{i-1})$ otherwise.

Proof. Verification of arity checksum for the pattern is a linear-time problem that will take $\mathcal{O}(m)$ time (see Algorithm 5.18).

Finding the occurrences of sub-pattern $p_i \neq \varepsilon$ takes time $\mathcal{O}(|p_i| + |occ(p_i)|)$. Summing over all sub-patterns yields total time $\mathcal{O}(m + \sum_{i=1, p_i \neq \varepsilon}^{k} |occ(p_i)|)$.

The merging time will be the sum of running times of all calls of Algorithm 5.21 with input size $\mathcal{O}(|occ(p_i)|)$, $p_i \neq \varepsilon$. Algorithm 5.21 outputs a list whose size is less than or equal to the minimum of the sizes of the two provided lists of occurrences. Thus, remembering that merging is not performed for $p_i = \varepsilon$, it must hold that the running time of all calls of Algorithm 5.21 will be less than or equal to $\mathcal{O}(\sum_{i=1, p_i \neq \varepsilon}^{k} (2 * |occ(p_i)|)) = \mathcal{O}(\sum_{i=1}^{k} |occ'(p_i)|)$.

Line 13 of the algorithm is called for every i to perform "jumps" on the prepared occurrences. The size of the prepared occurrences will not be greater than $|occ'(p_i)|$. □

5.4 Linear index as a simulation of a tree pattern PDA

A deterministic tree pattern PDA [50] supports index queries that run in time linear with respect to the pattern, but its size can be exponential with respect to the indexed tree (see Chapter 6 for details). The linear index presented in this chapter can be seen as an efficient simulation of a non-deterministic tree pattern PDA. The following is an outline of a proof, which shows that algorithm SearchPattern is a simulation of a non-deterministic tree pattern PDA (Algorithm 5.30).

Let t be a tree. Let $M_{npt}(t) = (Q, \mathcal{A}, G, \delta, 0, Z_0, \emptyset)$ be a non-deterministic tree pattern PDA for tree t. Let $pref(p) = p_1Sp_2S \dots Sp_k$ of total length m be the prefix notation of a tree pattern p, $p \in (\mathcal{A} \cup S)^*$. Let Q_{curr} denote the set of states that automaton $M_{npt}(t)$ is in at a single step of a computation. When reading $pref(p)$, the automaton performs the computation depicted in Algorithm 5.30. The description of the algorithm intentionally separates processing of sub-patterns from processing of S symbols so that comparison with the linear index from this chapter is easier.

The processing of sub-patterns performed by the pushdown automaton in Algorithm 5.30 is simulated by the linear index in algorithm SearchPattern, on lines 6 to 10. Before processing subpattern p_j, the pushdown automaton in Algorithm 5.30 is in a subset of states Q_{curr}. The algorithm FindOccurrences called by algorithm SearchPattern assumes the pushdown automaton is in the (full) set of states Q. Because of that, its output (set *occ*) has to be pruned of states that could not be reached from states in Q_{curr}. This is done by algorithm MergeOccurrences.

In Algorithm 5.30, after reading tree pattern prefix $p_1S \dots p_j$, automaton $M_{npt}(t)$ is simultaneously in states Q_{curr}. After reading the same input, algorithm SearchPattern maintains a set of occurrences *prevOcc* that was returned by MergeOccurrences.

At this moment in the algorithms, there exists a bijection between sets Q_{curr} and *prevOcc*. Set *prevOcc* can be converted to set Q_{curr} by reducing the occurrences to the elements $first$. Set Q_{curr} can be converted to *prevOcc* by adding a (unique) *last* element to each state q.

The processing of symbols S performed by the pushdown automaton in Algorithm 5.30 is simulated by the linear index in algorithm SearchPattern, on lines 11 to 13. Subtree jump table is used for storing the targets of the S transition rules. There again exists a bijection between the set of states Q_{curr} and the set of occurrences *prevOcc*.

Since *prevOcc* corresponds to Q_{curr} after processing any sequence $p_1Sp_2S \dots$, algorithm SearchPattern running on an index built for a tree t can be seen as an efficient simulation of a nondeterministic tree pattern PDA $M_{npt}(t)$.

Algorithm 5.30: Algorithm describing the computation of a non-deterministic tree pattern PDA $M_{npt}(t)$ on input $pref(t)$; pushdown store operations are omitted for clarity

```
Name: ReadPattern
Input: Tree pattern $p$, $pref(p) = p_1 S p_2 S \dots p_k$; $M_{npt}(t) = (Q, \mathcal{A}, G, \delta, 0, Z_0, \emptyset)$
Output: Ends of occurrences of tree pattern $p$ in tree $t$ stored in $Q_{curr}$
begin
    $Q_{curr} = \{0\}$;
    for $i := 1$ to $k$ do
        // process pattern $p_i$
        for $j := 1$ to $h$, where $p_i = a_1\ a_2 \dots a_h$ do
            $Q_{new} = \emptyset$;
            foreach $q \in Q_{curr}$ (do in parallel) do
                $Q_{new} = Q_{new} \cup \{q'\}$ for all $q'$, where $\delta(q, a_j, S) \ni (q', S^{Arity(a_j)})$;
            end
            $Q_{curr} = Q_{new}$;
        end
        // process symbol $S$
        if $i \neq k$ then
            $Q_{new} = \emptyset$;
            foreach $q \in Q_{curr}$ (do in parallel) do
                $Q_{new} = Q_{new} \cup \{q'\}$ for all $q'$, where $\delta(q, S, S) \ni (q', \varepsilon)$;
            end
            $Q_{curr} = Q_{new}$;
        end
    end
end
```

Chapter 6

On Space Requirements of Indexes of a Tree for Tree Patterns Based on FTA and on Tree Pattern PDA

This chapter deals with full indexes of a tree for tree patterns based on deterministic finite tree automaton (DFTA) and on deterministic tree pattern pushdown automaton (M_{dpt}). These automata can serve as indexes that allow to process input queries in linear time. However, as this section proves, they require exponential space for some trees. The contents of this section have been submitted for publication [A.3].

Finite tree automaton is a computational model that can be used for building an index of a tree for tree patterns and for tree pattern searching. A specific instance of a finite tree automaton for tree indexing is presented in the first section of this chapter. It has a deterministic and a non-deterministic variant. Its space requirements are analyzed.

A pushdown automaton specifically designed for tree pattern searching, a tree pattern pushdown automaton, has been proposed in [50]. It has a deterministic and a non-deterministic variant too. Its space requirements are analyzed in the second section.

The space complexity for the construction of the index of these two implementations is analyzed. It is proved that the minimal *deterministic tree pattern pushdown automaton* has a worst-case exponential size $\mathcal{O}(2^{n/4})$ for some indexed trees (an improvement over the square-root-exponential size $\mathcal{O}(2^{\sqrt{n}})$ proved in [29]). A similar exponential result is proved for a specific finite tree automaton. Moreover, a tree t with n nodes, $n = 4k+1, k \geq 1$, is introduced such that any *deterministic finite tree automaton* that accepts all tree patterns that match t (at any of its nodes) has at least $\mathcal{O}(2^{n/4})$ states. In other words - for some trees, a minimum deterministic finite tree automaton has exponential size when used as an index of a tree for tree patterns.

6.1 DFTA as an index for tree patterns

The major publications on finite tree automata [18, 35] do not directly present a finite tree automaton that would serve as a (full) index for tree patterns, i.e. which would accept all tree patterns that match a given tree. The construction of such finite tree automaton is straightforward, however. Algorithm 6.2 describes algorithm NFTA-for-tree-patterns, which creates a non-deterministic finite tree automaton as an index for tree patterns. Determinisation of this automaton is possible. An algorithm for construction of an equivalent deterministic FTA for a given NFTA is presented in [18] and is repeated here for clarity, because it will be used by subsequent algorithms.

Algorithm 6.1: Construction of equivalent deterministic finite tree automaton for a given non-deterministic finite tree automaton [18]

```
   Name: DET (for NFTA)
   Input: Non-deterministic FTA $M_N = (Q_N, \mathcal{A}, Q_{Nf}, \delta_N)$
   Output: Deterministic FTA $M_D = (Q_D, \mathcal{A}, Q_{Df}, \delta_D)$ equivalent with $M_N$
1  begin
2      Set $Q_D := \emptyset$;
3      Set $\delta_D := \emptyset$;
4      repeat
5          Set $Q_D := Q_D \cup \{s\}$;
6          Set $\delta_D := \delta_D \cup \{f(s_1, \ldots, s_n) \rightarrow s\}$;
           where: $f \in \mathcal{A}_n$, $s_1, \ldots, s_n \in Q_D$;
                  $s = \{q \in Q_N | \exists q_1 \in s_1, \ldots, q_n \in s_n, f(q_1, \ldots, q_n) \rightarrow q \in \delta_N\}$;
7      until no transition rule can be added to $\delta_D$;
8      Set $Q_{Df} := \{s \in Q_D | s \cap Q_{Nf} \neq \emptyset\}$;
9      return DFTA $M_D = (Q_D, \mathcal{A}, Q_{Df}, \delta_D)$;
10 end
```

When the NFTA automaton constructed by Algorithm 6.2 (NFTA-for-tree-patterns) encounters a symbol S while processing a tree pattern, it non-deterministically chooses the best subtree that fits in the place of the symbol S. It will be shown that when this automaton is determinised to a minimum equivalent DFTA, it can reach an exponential number of states with respect to the size of the input tree n.

Example 6.3 Consider tree t_4, $pref(t_4) = a4\ a_4\ a_4\ a_4\ a_0\ b_0\ a_0\ a_0\ a_0\ b_0\ a_0\ a_0\ a_0\ b_0\ b_0\ a_0\ a_0$. Tree t_4 is illustrated in Figure 6.1. Tree t_4 is an example of a tree for which minimum deterministic finite tree automata have an exponential number of states with respect to the number of states of the equivalent NFTA constructed by Algorithm 6.2 (NFTA-for-tree-patterns). For this tree, the minimum deterministic tree pattern PDAs also have an exponential number of states (see theorems 6.10 and 6.13).

Algorithm 6.2: Construction of non-deterministic finite tree automaton that accepts all tree patterns that match a tree t

```
   Name: NFTA-for-tree-patterns
   Input: Tree $t$, $pref(t) = a_1 \dots a_n$, $a_1, \dots, a_n \in \mathcal{A}$
   Output: Non-deterministic FTA $M_N$ that accepts all tree patterns over $\mathcal{A} \cup \{S\}$
           that match tree $t$
1  begin
2      construct a deterministic FTA 6.1 $M_N = (Q, \mathcal{A}, Q_f, \delta)$ that accepts only tree t
         and its subtrees;
3      foreach transition rule $r = a(q_1(x_1)q_2(x_2) \dots q_k(x_k)) \to q(a(x_1 x_2 \dots x_k))$, $r \in \delta$,
         $k \geq 0$ do
4          insert into $\delta$ a new transition rule $S \to q$;
5      end
6  end
```

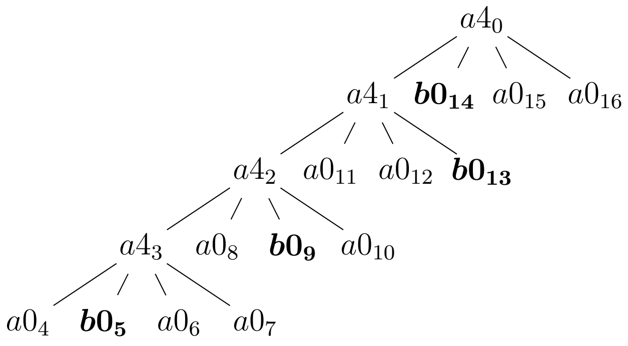


Figure 6.1: Tree t_4 from Example 6.3

Example 6.4 Let alphabet $\mathcal{A} = \{a4, a0, b0\}$. Consider DFTA $M = (Q, \mathcal{A}, Q_f, \delta)$, where $Q = \{q_a, q_b, q_1, q_2, q_3, q_4\}, Qf = Q$, and

$$\begin{array}{rcl} \delta = \{\ a0 & \to & q_a(a0), \\ b0 & \to & \boldsymbol{q_b}(b0), \\ a4(q_a(x_1)\ \boldsymbol{q_b}(x_2)\ q_a(x_3)\ q_a(x_4)) & \to & q_1(a4(x_1\ x_2\ x_3\ x_4)), \\ a4(q_1(x_1)\ q_a(x_2)\ \boldsymbol{q_b}(x_3)\ q_a(x_4)) & \to & q_2(a4(x_1\ x_2\ x_3\ x_4)), \\ a4(q_2(x_1)\ q_a(x_2)\ q_a(x_3)\ \boldsymbol{q_b}(x_4)) & \to & q_3(a4(x_1\ x_2\ x_3\ x_4)), \\ a4(q_3(x_1)\ \boldsymbol{q_b}(x_2)\ q_a(x_3)\ q_a(x_4)) & \to & q_4(a4(x_1\ x_2\ x_3\ x_4)),\ \}. \end{array}$$

This finite tree automaton is a minimum DFTA [18] that accepts tree t_4 from Example 6.3 and all its subtrees. It has 6 states and 6 transition rules.

Example 6.5 Let $\mathcal{A} = \{a4, a0, b0\}$. Consider NFTA $M_N = (Q, \mathcal{A} \cup \{S\}, Q_f, \delta)$, where $Q = \{q_a, q_b, q_1, q_2, q_3, q_4\}, Q_f = Q$, and

$$\begin{array}{rcl} \delta = \{\ a0 & \to & q_a(a0), \\ b0 & \to & \boldsymbol{q_b}(b0), \\ a4(q_a(x_1)\ \boldsymbol{q_b}(x_2)\ q_a(x_3)\ q_a(x_4)) & \to & q_1(a4(x_1\ x_2\ x_3\ x_4)), \\ a4(q_1(x_1)\ q_a(x_2)\ \boldsymbol{q_b}(x_3)\ q_a(x_4)) & \to & q_2(a4(x_1\ x_2\ x_3\ x_4)), \\ a4(q_2(x_1)\ q_a(x_2)\ q_a(x_3)\ \boldsymbol{q_b}(x_4)) & \to & q_3(a4(x_1\ x_2\ x_3\ x_4)), \\ a4(q_3(x_1)\ \boldsymbol{q_b}(x_2)\ q_a(x_3)\ q_a(x_4)) & \to & q_4(a4(x_1\ x_2\ x_3\ x_4)), \\ S & \to & q_a(S), \\ S & \to & \boldsymbol{q_b}(S), \\ S & \to & q_1(S), \\ S & \to & q_2(S), \\ S & \to & q_3(S)\ \ \}. \end{array}$$

This finite tree automaton, constructed using Algorithm 6.2 (NFTA-for-tree-patterns) is an NFTA that accepts tree t_4 from Example 6.3, all its subtrees and all tree patterns that match tree t_4. It has 6 states and 10 transition rules.

Note that NFTA M_N accepts a single symbol S. It would be possible to modify this NFTA to not accept a single symbol S, but the size of the NFTA rises substantially. The modification requires that $Q' = \{q'_a, q'_b, q'_1, q'_2, q'_3\}$, $Q = Q \cup Q'$ and transition rules $S \to q_a(S), \ldots, S \to q_3(S)$ be replaced with transition rules $S \to q'_a(S), \ldots, S \to q'_3(S)$. For each transition rule of form $a4(\ldots) \to q(a4(\ldots))$, 2^5 new transition rules must be added to accommodate for new states from Q'.

Using determinisation algorithm DET for NFTAs [18], NFTA M_N can be transformed into an equivalent DFTA M_D. The deterministic FTA is shown in the next example.

Example 6.6 Deterministic FTA M_D constructed by determinisation algorithm DET [18] from FTA M_N from Example 6.5 (that accepts tree t_4 from Example 6.3, all its subtrees and all tree patterns that match tree t_4) has 18 states and more than 100 transition rules. $M_D = (Q_d, \mathcal{A} \cup \{S\}, Q_{fd}, \delta_d)$, where $Q_d = \{q_a, q_b, q_1, q_2, q_3, q_4, q_{ab123}, q_{12}, q_{13}, q_{23}, q_{14}, q_{24}, q_{34}, q_{123}, q_{124}, q_{134}, q_{234}, q_{1234}\}$, $Q_{fd} = Q$. Part of transition function δ_d of the automaton is illustrated in the following table:

$$\begin{array}{rcl}
\delta_d = \{ \ a0 & \to & q_a(a0), \\
b0 & \to & q_b(b0), \\
a4(q_a(x_1)\ q_b(x_2)\ q_a(x_3)\ q_a(x_4)) & \to & q_1(a4(x_1\ x_2\ x_3\ x_4)), \\
a4(q_1(x_1)\ q_a(x_2)\ q_b(x_3)\ q_a(x_4)) & \to & q_2(a4(x_1\ x_2\ x_3\ x_4)), \\
\ldots, & & \\
a4(q_{ab123}(x_1)\ q_b(x_2)\ q_a(x_3)\ q_a(x_4)) & \to & q_{14}(a4(x_1\ x_2\ x_3\ x_4)), \\
a4(q_{ab123}(x_1)\ q_a(x_2)\ q_b(x_3)\ q_a(x_4)) & \to & q_2(a4(x_1\ x_2\ x_3\ x_4)), \\
\ldots, & & \\
a4(q_a(x_1)\ q_{ab123}(x_2)\ q_a(x_3)\ q_a(x_4)) & \to & q_1(a4(x_1\ x_2\ x_3\ x_4)), \\
\ldots, & & \\
a4(q_{ab123}(x_1)\ q_{ab123}(x_2)\ q_a(x_3)\ q_a(x_4)) & \to & q_{14}(a4(x_1\ x_2\ x_3\ x_4)), \\
\ldots, & & \\
a4(q_{ab123}(x_1)\ q_{ab123}(x_2)\ q_{ab123}(x_3)\ q_a(x_4)) & \to & q_{124}(a4(x_1\ x_2\ x_3\ x_4)), \\
a4(q_{ab123}(x_1)\ q_{ab123}(x_2)\ q_a(x_3)\ q_{ab123}(x_4)) & \to & q_{134}(a4(x_1\ x_2\ x_3\ x_4)), \\
a4(q_{ab123}(x_1)\ q_a(x_2)\ q_{ab123}(x_3)\ q_{ab123}(x_4)) & \to & q_{23}(a4(x_1\ x_2\ x_3\ x_4)), \\
a4(q_{ab123}(x_1)\ q_{ab123}(x_2)\ q_{ab123}(x_3)\ q_{ab123}(x_4)) & \to & q_{1234}(a4(x_1\ x_2\ x_3\ x_4)), \\
\ldots, & & \\
a4(q_{123}(x_1)\ q_{ab123}(x_2)\ q_{ab123}(x_3)\ q_{ab123}(x_4)) & \to & q_{234}(a4(x_1\ x_2\ x_3\ x_4)), \\
S & \to & q_{ab123}(S)\ \ \}.
\end{array}$$

This FTA is not minimum, because any its state of the form $q_R, R \subseteq \{a, b, 1, 2, 3, 4\}, 4 \in R, |R| > 1$, is equivalent to state $q_{(R)\setminus\{4\}}$. In the minimum FTA, state q_{14} is equivalent to state q_1, state q_{234} is equivalent to state q_23 etc.

Note that the deterministic FTA has new states $q_{12}, q_{13}, q_{23}, q_{123}$ etc., which represent all possible combination of states q_1, q_2, q_3, q_4 of the original non-deterministic FTA. With increasing arity of the non-leaf symbols, the number of these combinations rises exponentially with respect to arity r of the non-leaf symbols. Even in the minimum DFTA the number of states rises exponentially, as is shown by Theorem 6.10.

The following two definitions are needed for the proof that the minimum deterministic FTA can have an exponential number of states w.r.t the tree t for which the DFTA serves as an index for tree patterns. The definitions are based on [18].

Definition 6.7 Let τ be a set of all trees over alphabet $\mathcal{A}$. An equivalence relation $\equiv$ on τ is a *congruence* $\cong$ on τ if for every $a \in \mathcal{A}_n$

$$(\forall\ 1 \leq i \leq n, u_i \equiv v_i) \Rightarrow a(u_1 \ldots u_n) \equiv a(v_1 \ldots v_n).$$

The congruence is of finite index if there are finitely many equivalence classes.

Definition 6.8 Let $\mathcal{X}$ be a set of variables. Let τ be a set of all trees over alphabet $\mathcal{A}$. Let L be a tree language over $\mathcal{A}$. Congruence $\cong_L$ on τ is defined by:

$$u \cong_L v \text{ if for all contexts } C \text{ over } \mathcal{A} \cup \mathcal{X}, C[u] \in L \Leftrightarrow C[v] \in L.$$

The following definition of distinguishing context is similar to the notion of distinguishing suffix as used by the Myhill-Nerode theorem for regular languages [38].

Definition 6.9 Let $\mathcal{X}$ be a set of variables. Let L be a tree language over $\mathcal{A}$. Let u, v be trees such that $u \not\cong_L v$. Then there must exist a context C over $\mathcal{A} \cup \mathcal{X}$ such that ($C[u] \in L$ and $C[v] \notin L$) or ($C[u] \notin L$ and $C[v] \in L$). Context C is called a *distinguishing context* for trees u, v in L.

The following theorem uses a generalized version of automaton M_N from Example 6.5. Whereas M_N accepts a tree whose nodes are either of arity 0 or 4, the generalized automaton accepts trees of the same kind whose nodes are either of arity 0 or $k+1$ for $k > 0$.

The theorem itself exploits an already known fact that the determinisation process of NFTA can result in an exponential blow-up of states [18]. Its proof shows a non-deterministic FTA with $k+2$ states, $k > 0$ for which the minimum complete equivalent deterministic FTA has $2^k + 2$ states.

Theorem 6.10 Let N_t^S denote an NFTA that accepts all tree patterns that match a tree t. Let D_t^S denote a minimum complete DFTA equivalent to N_t^S. Then there is a tree te_k with $n = (k+1) * (k+1) + 1$ nodes, $k > 0$, for which there exists an NFTA $N_{te_k}^S$ with $k+3 = \mathcal{O}(\sqrt{n})$ states and for which the minimum complete DFTA $D_{te_k}^S$ has $2^k+3 = \mathcal{O}(2^{\sqrt{n}})$ states.

Proof. Let $\mathcal{A} = \{a0, b0, ak\}$ be an alphabet. Let M_{ne} be a non-deterministic FTA with $k+2$ states and the following structure: $M_{ne} = (Q, \mathcal{A} \cup \{S\}, Q_f, \delta)$, $Q = \{q_a, q_b, q_1, q_2, \ldots, q_k, q_{k+1}\}$, $Q_f = \{q_1, q_2, \ldots, q_{k+1}\}$ and

$$\begin{array}{rcl}
\delta = \{ \; a0 & \rightarrow & q_a(a0), \\
b0 & \rightarrow & \boldsymbol{q_b}(b0), \\
ak(q_a(x_1)\ \boldsymbol{q_b}(x_2)\ q_a(x_3)\ \ldots\ q_a(x_{k+1})) & \rightarrow & q_1(ak(x_1\ x_2\ x_3\ \ldots\ x_{k+1})), \\
ak(q_1(x_1)\ q_a(x_2)\ \boldsymbol{q_b}(x_3)\ \ldots\ q_a(x_{k+1})) & \rightarrow & q_2(ak(x_1\ x_2\ x_3\ \ldots\ x_{k+1})), \\
\ldots & & \\
ak(q_{k-1}(x_1)\ q_a(x_2)\ q_a(x_3)\ \ldots\ \boldsymbol{q_b}(x_{k+1})) & \rightarrow & q_k(ak(x_1\ x_2\ x_3\ \ldots\ x_{k+1})), \\
ak(q_k(x_1)\ \boldsymbol{q_b}(x_2)\ q_a(x_3)\ \ldots\ q_a(x_{k+1})) & \rightarrow & q_{k+1}(ak(x_1\ x_2\ x_3\ \ldots\ x_{k+1})), \\
S & \rightarrow & q_a(S), \\
S & \rightarrow & \boldsymbol{q_b}(S), \\
S & \rightarrow & q_1(S), \\
S & \rightarrow & q_2(S) \;\}.
\end{array}$$

Given a specific value of k, the automaton M_{ne} accepts a certain tree te_k and all tree patterns that match tree te_k. An example of tree te_k for $k = 3$ is shown in Figure 6.1. The construction used for creating automaton M_{ne} is analogous to the construction of NFTA for tree t_4 from Example 6.5.

1. Using determinisation of NFTA M_{ne} by algorithm DET (6.1), an equivalent DFTA $M_{de} = (Q_d, \mathcal{A} \cup S, Q_{fd}, \delta_d)$ is created. By application of DET, it holds that $Q_d = \{q_a, q_b\} \cup P$ and $Q_{fd} = P$, where $P = \{q_p | p \in 2^K$, where $K = \{1, 2, \ldots, k\}\}$.

2. Consider the minimisation of DFTA M_{de}. It is obvious that states q_a, q_b lie in their own equivalence classes.

   Consider number $i, 1 \leq i \leq k$, [1] and any state q_p from P, $i \in p$. There exists a tree pattern te_p over $\mathcal{A} \cup S$ such that $te_p \underset{M_D}{\overset{*}{\rightarrow}} q_p(te_p)$ and there exists a tree te_i over $\mathcal{A}$ such that $te_i \underset{M_D}{\overset{*}{\rightarrow}} q_{\{i\}}(te_i)$. There is at least one tree context C_i over $\mathcal{A}$ such that $C_i[te_i]$ is a tree accepted by M_{ne} and $C_i[te_p]$ is a tree pattern accepted by M_{ne}.

   Consider, without loss of generality, state q_r from P, $q_r \neq q_p$, $i \notin r$. There exists a tree pattern te_r such that $te_r \underset{M_D}{\overset{*}{\rightarrow}} q_r(te_r)$. Because of determinisation, being in state q_r is equivalent to being in all states $q_{\{j\}}$, $j \in r$, at once. Since $q_{\{j\}} \neq q_{\{i\}}$ for all $j \in r$ and $q_{\{i\}}$ has a unique distinguishing context for all $i \leq k$, context C_i is a distinguishing context for tree patterns te_r and te_p and thus $te_r \not\cong_{L(M)} te_p$. That means that states q_r and q_p are not equivalent.

   If the state $q_{\{\}}$ is considered as the error state, then the number of states of the minimum complete DFTA M_{de} is $|Q_d| = 2^k + 3$.

   Since automata M_{ne} and M_{de} are equivalent and they both accept tree te_k and all its subtrees, this concludes the proof.

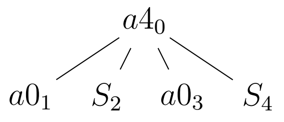
□

Expressed informally, consider tree t_4 from Example 6.3 and generalize its depth and arity of non-constant nodes by k. Take any two different sets of nodes A and A' of arity k, $A \subseteq \{ak_1, ak_2, \ldots ak_k\}$, $A' \subseteq \{ak_1, ak_2, \ldots ak_k\}$. The proof notices that one can construct a tree pattern that matches the subject tree at (and only at) all nodes from A. Such pattern has a symbol S where any node $ak \in A$ has a node $b0$ as a direct descendant. For every subtree te_i rooted at node ak_i, $1 \leq i \leq k$, there exists a unique tree context C_i such that $C_i[te_i]$ is accepted by M_{ne} and $C_i[te_i]$ contains the root node of generalized version of tree t_4. Therefore any two different sets of nodes A and A' have a distinguishing context and cannot be represented by the same state.

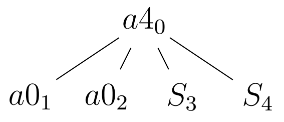
Corollary 6.11 For some non-deterministic finite automata that accept all tree patterns that match a given tree, the equivalent *minimum* determinstic finite automata have exponentially more states.

Proof. The proof directly follows from the proof of Theorem 6.10. □

[1]not $\leq k + 1$ due to technical reasons



Pattern p_{13}, $pref(p_{13}) = a4\ a0\ S\ a0\ S$



Pattern p_{23}, $pref(p_{23}) = a4\ a0\ a0\ S\ S$

Figure 6.2: Patterns p_{13} and p_{23} that match tree t_4 from Example 6.3

Example 6.12 Consider tree t_4 from Example 6.3. This tree is the tree from Theorem 6.10 with $k = 3$. The minimum deterministic FTA $M_{t_4}^S$ that accepts tree t_4 and all patterns that match it has states $Q_d = \{q_a, q_b, q_1, q_2, q_3, q_4, q_{12}, q_{23}, q_{13}, q_{14}, q_{24}, q_{34}, \ldots\}$. Its final states are $Q_{fd} = Q_d \setminus \{q_a, q_b\}$. Two patterns p_{13} and p_{23} that cannot lead DFTA $M_{t_4}^S$ to the same final state are shown in Figure 6.2. After processing patterns p_{13} and p_{23} from input, DFTA $M_{t_4}^S$ ends in states q_{13} and q_{23}, respectively. Note that although both states are final, they are not equivalent. Two states q_a, q_b of an FTA are equivalent if for any single-variable tree context C the trees $C[q_a]$ and $C[q_b]$ are either both accepted by the FTA or both rejected by the FTA [18].

6.2 Deterministic tree pattern PDA as an index for tree patterns

The tree pattern pushdown automaton [50] is a specific pushdown automaton that accepts all tree patterns that match a given tree or its subtrees. It was designed to serve as an index for tree patterns. In non-deterministic form, it takes linear space with respect to the size of the indexed tree. After determinisation, however, the automaton can have exponential size. An example of a tree for which the deterministic tree pattern PDA has exponential number of states is shown in Example 6.3. Notice that the finite tree automaton that accepts tree patterns that match this tree has exponential size too.

The Theorem 6.10 can be extended to the case of tree pattern PDA. The proof uses a generalized version of tree t_4 from Example 6.3. The general form of tree t_4 used as t_k in the proof is shown in Figure 6.3. Note that it was already shown that a deterministic tree pattern PDA can have exponential size $\mathcal{O}(2^{\sqrt{n}})$ in [29]. The proof shown here is interesting because it uses the same tree as the tree shown in the proof of the exponential size of deterministic finite tree automata. Moreover, Subsection 6.3.1 shows a stronger result: a

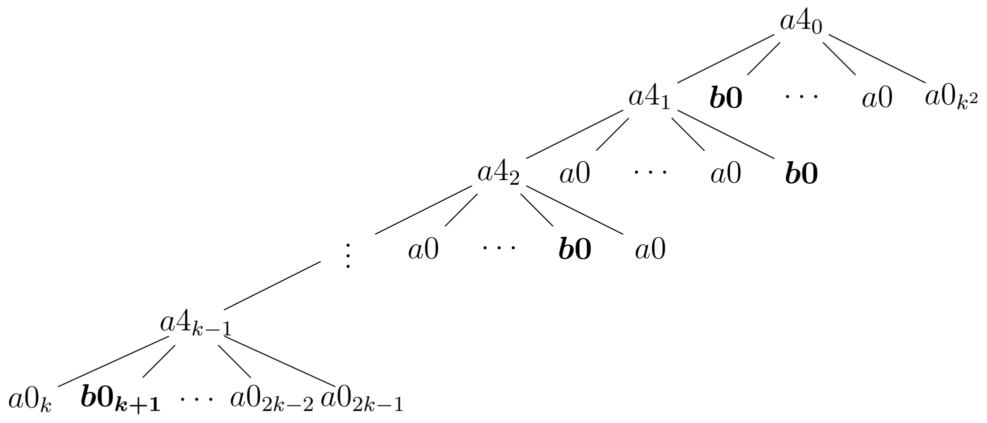


Figure 6.3: Tree t_k from the proof of Theorem 6.13; indexes of some leaf symbols are omitted for the sake of clarity

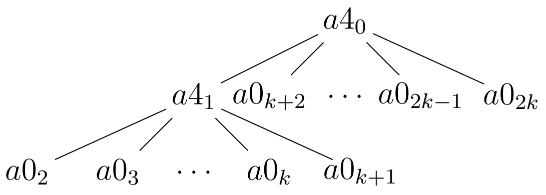


Figure 6.4: A "template" for tree patterns from set p of tree patterns used in the proof of Theorem 6.13

minimal deterministic tree pattern PDA of size $\mathcal{O}(2^{n/4})$.

Theorem 6.13 Let t be a tree over alphabet $\mathcal{A}$. Let $M_{npt}(t)$ denote a non-deterministic tree pattern PDA that accepts tree t and all tree patterns that match t. Let $M_{dpt}(t)$ denote a deterministic tree pattern PDA equivalent to $M_{npt}(t)$. Then there is a tree t_k with $n = k^2 + 1$ nodes, whose automaton $M_{npt}(t_k)$ has $n + 1 = k^2 + 2$ states and whose automaton $M_{dpt}(t)$ cannot have less than $2^k = \mathcal{O}(2^{\sqrt{n}})$ states.

Proof. Let k be an integer, $k > 0$. Let t_k be a tree with $k^2 + 1$ nodes. The structure of tree t_k is illustrated in Figure 6.3. The non-deterministic tree pattern PDA $M_{npt}(t_k)$ that accepts tree t_k and all patterns that match it constructed by Algorithm 2.21 from Chapter 2 has $k^2 + 2$ states. The equivalent deterministic tree pattern PDA $M_{dpt}(t_k)$ constructed by Algorithm 2.23 from Chapter 2 has at least 2^k states, but it is not minimal. The following steps show that the minimal tree pattern automaton cannot have less than 2^k states.

1. The minimal automaton must have an initial state.
2. Consider a set of tree patterns $p = \{p_K | K \in 2^{\{2,3,\ldots,k+1\}}\}$. Each pattern p_K, $p_K \in p$ is of the form $pref(p_K) = a4_0\ a4_1\ a0_2\ a0_3\ \ldots a0_{2k}$, where symbol $a0$ at position i is

replaced by symbol S if $i \in K$. This "template" for tree patterns from p is illustrated in Figure 6.4. The set p contains all tree patterns that match this template and do not contain a symbol S at position 1. All these patterns match tree t_k.

3. Consider any two different tree patterns p_1, p_2 from set P. After reading the first $k+2$ symbols of patterns p_1 and p_2, the deterministic tree pattern PDA $M_{dpt}(t_k)$ ends in two different states q_1, q_2, respectively. States q_1 and q_2 are not equal because of the same reason as in the proof of Theorem 6.10. There must exist a number i that belongs to p_1 and not to p_2 (or vice versa). This number identifies a unique distinguishing suffix *suff* such that $\mathit{pref}(p_1)[0]\mathit{pref}(p_1)[1]\ldots\mathit{pref}(p_1)[k+1]\mathit{suff}$ is a prefix notation of a tree pattern accepted by $M_{dpt}(t_k)$ and $\mathit{pref}(p_2)[0]\mathit{pref}(p_2)[1]\ldots\mathit{pref}(p_2)[k+1]\mathit{suff}$ is a prefix notation of a tree pattern not accepted by $M_{dpt}(t_k)$ (or vice versa).

   Thus states q_1 and q_2 cannot be equal. This means that after reading the first $k+2$ symbols of the tree pattern, the minimal automaton can end up in more than 2^{k-1} different states.

4. Consider tree patterns from the set of tree patterns P. In their prefix notation, the tree patterns have either a symbol S or an $a0$ symbol on positions 2 and beyond. After reading the first two symbols of any of tree patterns P, the minimal automaton ends in the same state $q_{\{2,3,\ldots,k+1\}_1}$. After reading the next symbol ($a0$ or S), the automaton can end in two new different states: $q_{\{2,3,\ldots,k+1\}_2}$ or $q_{\{3,\ldots,k+1\}_2}$. After reading the next symbol, the automaton can end in four new different states: $q_{\{2,3,\ldots,k+1\}_3}$, $q_{\{2,\ldots,k+1\}_3}$, $q_{\{3,\ldots,k+1\}_3}$ or $q_{\{4,\ldots,k+1\}_3}$. Eventually, after reading $k+2$ symbols, the automaton can end in 2^{k-1} new different states. In total, this is $2^k - 1$ states.

5. Parsing only patterns from the set of tree patterns P, the minimal automaton can visit up to $2^k - 1$ different states, plus the initial state. Thus the minimal automaton cannot have less than 2^k states.

□

Note that the tree t_k from the proof of Theorem 6.13 is only slightly different from the tree t_4, whose generalized form was used in the proof of Theorem 6.10. Tree t_k from the proof of Theorem 6.13 could in fact be used for both proofs without (more than a constant) effect on the exponential size result.

6.3 Trees with small indexes, trees with large indexes

In the previous sections, trees with exponential-size indexes have been shown. One may then ask the following question - is there a property that trees with exponential-size indexes share? And on the contrary, what property do trees with sub-exponential indexes have?

A conclusive answer is not provided here. However, several properties are identified and three trees, two of them with index of size $\mathcal{O}(n)$, another with index of size $\mathcal{O}(2^{n/4})$, are described. The case of tree pattern PDA is considered. The case of Finite Tree Automaton is analogous.

6.3.1 Upper bound on the number of states of the index

6.3.1.1 Number of different subtrees of a tree

Consider a tree t indexed by a deterministic tree pattern PDA M_{dpt}. Consider two identical subtrees t_1, t_2 of tree t. Such subtrees cannot be distinguished by tree pattern PDA M_{dpt}. For instance, after reading any prefix of tree t_1, the PDA will end in state q. After reading the same prefix of tree t_2, it must end in the same state q. Since the index of tree t does not distinguish between identical subtrees, its size must be determined by the number of different subtrees of tree t.

There are types of trees that have very few different subtrees. For instance, a complete binary tree (labeled by the same label on all nodes) has $n = 2^k - 1$ nodes, but it has only k different subtrees. The maximum possible number of different d-subsets is bounded by the number of possible combinations of these k different subtrees, multiplied by at most their length (see Lemma 6.14 for details). There are 2^k possible combinations and the length is at most $2^k - 1$. Therefore the index (tree pattern PDA M_{dpt}) cannot have more than $2^k * (2^k - 1) < (n + 1)^2$ nodes. But the upper bound in this case lies even lower - the minimal deterministic tree pattern PDA for a complete binary tree with n nodes has only $n + 1$ states - many states are merged together during minimisation. The reason for this is the fact that the suffixes of tree patterns accepted by the PDA M_{dpt} are suffixes of tree patterns that match tree t at its root node. This allows many states to be merged together. The resulting automaton strongly resembles treetop PDA for t, which has $n + 1$ states.

Lemma 6.14 Assume two trees t_1, t_2 whose roots are of the same arity and are labeled by the same symbol. Assume a tree pattern p that matches them both at the root node and has the most non-S nodes out of all tree patterns that match the two trees at the root node. Tree pattern p implies a 1:1 mapping between nodes matched by its non-S symbols in the the two trees. Any other tree pattern that matches the two trees at the root node implies a subset of this mapping.

Proof. It is apparent that tree pattern p implies a mapping between the nodes of the two trees. See Figure 6.5 for an illustration of this property. The sizes of subtrees of t_1 and t_2 that are matched by symbols S of tree patttern p can be different, but are fixed. Any tree pattern p' different from p that still matches subtrees t_1, t_2 at the root node differs from p by replacing some of the subtrees of p by an S symbol. Take one subtree of p which was replaced by an S symbol; call this symbol S_1. The subtree contained some S symbols, which altogether skipped k_1 symbols in tree t_1 and k_2 symbols in tree t_2. It also contained

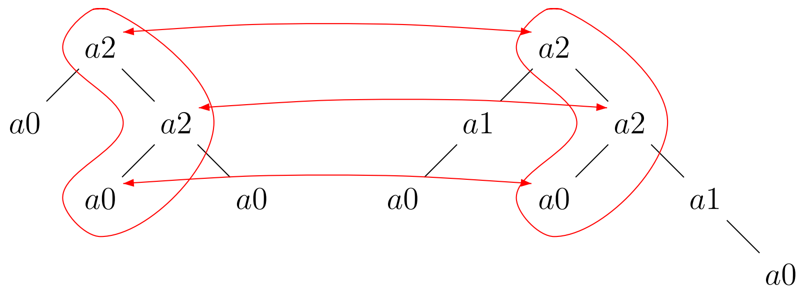


Figure 6.5: Mapping between nodes of two trees implied by tree patterns that match them

some non-S symbols, which skipped k_3 symbols in both trees. But $k_1 + k_3$ is the size of the subtree in t_1 matched by symbol S_1 and $k_2 + k_3$ is the size of the subtree in t_2 matched by symbol S_1. No relative shift of indexes in trees t_1 and t_2 occured. Therefore the mapping implied by tree pattern p' is consistent with (is subset of) the mapping implied by tree pattern p. $\square$

6.3.1.2 Distinguishing suffixes

Recall that the proofs in the previous section used the notion of *distinguishing context* and *distinguishing suffix*. Consider a deterministic tree pattern PDA determinised using determinisation algorithm 2.23. Take any of its d-subsets d. The elements of d correspond to prefixes $pref_i$ of prefix notations of all subtrees $t_1, t_2, \ldots, t_k$ that are 'matched' by the same prefix of prefix notation of some tree pattern. We will say that two subtrees t_1, t_2 are *indistinguishable in d-subset* d if there exist identical suffixes $suff_1$ and $suff_2$, such that $pref_i suff_i = pref\_bar(t_i)$ for $i \in \{1, 2\}$.

In principle, inserting an S transition to a subtree PDA may cause some d-subsets (corresponding to states of the PDA) to keep their size upon reading the S symbol rather than splitting into smaller d-subsets. Inserting an S symbol may thus create new d-subsets in the automaton. If those new d-subsets contain indistinguishable subtrees, it may be possible to merge these new d-subsets with already existing d-subsets and the size of the PDA will not increase. The following examples show such trees.

6.3.1.3 Examples of very small and very large indexes

Example 6.15 Figure 6.6 provides an example of a tree whose tree pattern PDA has d-subsets that correspond to indistinguishable subtrees. Minimised tree pattern PDA constructed for this tree is illustrated in Figure 6.7. The number of states of this PDA is further reduced thanks to the fact that the PDA accepts tree patterns by empty pushdown store (for instance, d-subset $\{6\}$ could be merged with d-subset $\{5, 6\}$).

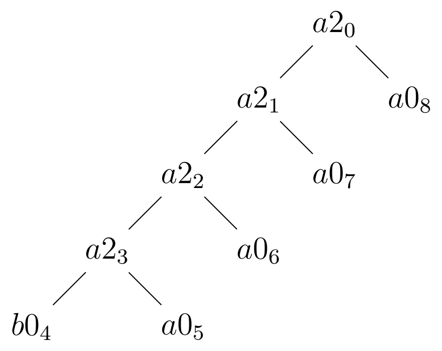


Figure 6.6: An example of a tree whose tree pattern PDA has a very low number of states thanks to a high number of indistinguishable subtrees in the d-subsets

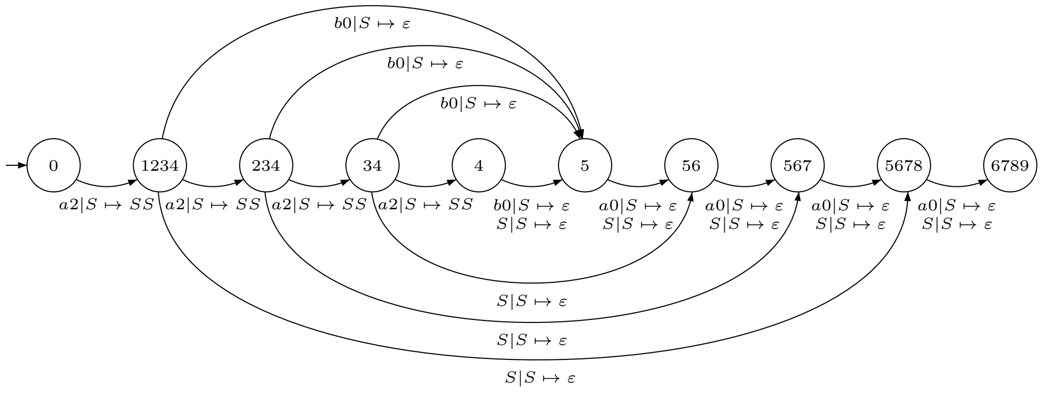

$b0|S \mapsto \varepsilon$
$b0|S \mapsto \varepsilon$
$b0|S \mapsto \varepsilon$
0
1234
234
34
4
5
56
567
5678
6789
$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a2|S \mapsto SS$
$b0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$

Figure 6.7: Minimal tree pattern PDA for tree from Figure 6.6

Example 6.16 Drawing on the notions of different subtrees and distinguishing suffixes, one may also construct trees for which the index size is exponential. Consider the tree t_e from Figure 6.8, $pref(t_e) = \boldsymbol{b2}\ a0\ a2\ a0\ a2\ a0\ \ldots a2\ a0\ a2\ \boldsymbol{b0}\ a2\ a0\ a2\ a0\ \ldots a2\ a0\ \boldsymbol{b2}\ a0\ a0$. All subtrees of tree t_e that begin with symbol a2 are different. Moreover, given any d-subset that corresponds to at least two different subtrees of t_e that begin with symbol $a2$, the subtrees have a distinguishing suffix and cannot be treated as the same subtree. Lastly, the d-subsets of tree pattern PDA for tree t_e can grow or shrink by one element at a time due to the structure of the tree. These three properties are enough to make the size of the index exponential with respect to the size of the indexed tree. The following lemma provides a precise statement.

Lemma 6.17 Let t_e be a tree with n nodes, $pref(t_e) = \boldsymbol{b2}\ a0\ a2\ a0\ a2\ a0\ \ldots a2\ a0\ a2\ \boldsymbol{b0}\ a2\ a0\ a2\ a0\ \ldots a2\ a0\ \boldsymbol{b2}\ a0\ a0$, where the symbol $b0$ occurs in the middle of the prefix notation of the tree. The minimal tree pattern PDA that indexes it cannot have less than $\mathcal{O}(2^{n/4})$ nodes. The minimum DFTA that indexes it also cannot have less than $\mathcal{O}(2^{n/4})$ nodes.

Proof. The proof to this lemma for tree pattern PDA and for DFTA is analogous to the proof of Theorem 6.13 and the proof of Theorem 6.10, respectively. The distinguishing contexts for the FTA proof are inserted from top, whereas the distinguishing suffixes for the tree pattern PDA proof are inserted from the bottom. Note that the number of nodes (or states) is $\mathcal{O}(2^{n/(2*2)})$. The number of nodes (or states) in this expression is twice devided by 2 because for each symbol of arity 0 there is one symbol of arity 2 (leaving n/2 symbols of arity 0) and because the tree patterns used in the proof contain up to k = one half of all symbols of arity 0 (second division). See the Figure 6.8 for a visual explanation. □

This is a stronger result than proved in Theorems 6.13 and 6.10.

6.3.2 Discussion

We show two examples of deterministic pushdown automata used as an index (of a tree for tree patterns). Both have worst-case exponential size. It is therefore natural to consider the question whether there is any deterministic pushdown automaton that could be used as an index and have less than exponential size. The grammar that generates the language L_P of all tree patterns that match a tree t in linear prefix form has linear size with respect to the number of nodes of tree t (the rules can be in Greibach normal form). The theory of deterministic context-free languages shows that there are languages whose parsers (based on deterministic PDAs) have size exponential compared to the generating grammars [59]. It is possible that language L_P is such a language. Language L_P is finite. From the results presented in this text it is obvious that it cannot be accepted by a deterministic finite automaton of less than exponential size. It is unknown whether it is possible to build a parser of language L_P that would run in linear time and have linear size. If so, the computational model is unclear as well. The deterministic PDA does not seem suitable.

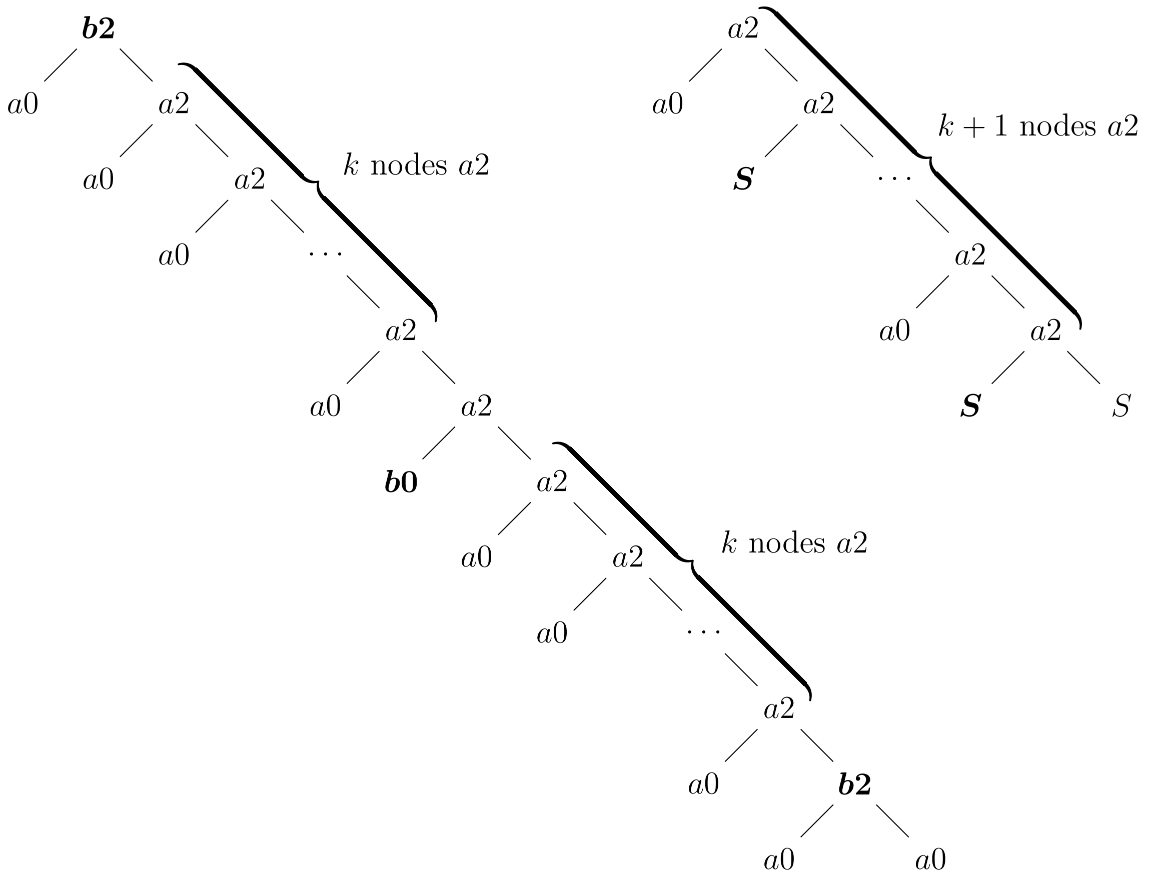


Figure 6.8: Tree t_e whose tree pattern PDA has an exponential number of states $\mathcal{O}(2^{n/4})$ and an example of a tree pattern that matches two of its subtrees

The next step in the computational complexity hierarchy, the class of growing context-sensitive grammars, recognizable by length-reducing two-pushdown automata, could be the candidate for such parser. It has the desirable property that the membership problem is still decidable in polynomial time [42]. Computational model of a Turing Machine has been successfully used for building a linear-size index, but its matching time was super-linear [A.2].

6.3.3 Conclusion

It has been shown that there exists a tree such that any deterministic finite tree automaton that accepts all tree patterns that match this tree has an exponential number of states $\mathcal{O}(2^{n/4})$ with respect to the number of nodes n of the tree. The same result was proved for the case of the deterministic tree pattern pushdown automaton (an improvement over the result in [29]). This seems to suggest that if one wishes to have an index of a tree for tree patterns such that the searching phase takes linear time, the index size is at worst exponential, at least under the computational model of pushdown automaton. Given that the index of a tree for tree patterns presented in Section 5.1 of Chapter 5 has linear size, it is of no surprise that the complexity of the searching phase of this index is not strictly linear with respect to the size of the indexed tree.

Chapter 7

Conclusions

This doctoral thesis set three goals. Firstly, to propose an index for exact tree indexing that can be used for compression of the indexed tree. Secondly, to explore and improve current methods of indexing of a tree for tree patterns. Thirdly, to investigate the time and space requirements for the construction of an index of a tree for tree patterns based on finite tree automaton and on tree pattern pushdown automaton. The results were put into context with the pushdown automaton approach coined by arbology [3].

7.1 Tree compression automaton

In Chapter 4, a new method for tree indexing was proposed. A kind of pushdown automaton called tree compression automaton (TCA) was introduced. The TCA is a special version of a general tree compression automaton (GTCA), which was defined to accept by empty pushdown store all subtrees in prefix bar notation [3] of trees in a given set T. The TCA is a GTCA that was constructed by Algorithm 4.12 (TCA-construction). The TCA was proved to be deterministic in Theorem 4.14.

It was proved that TCA can be used as an index for not only a single tree, but for a set of trees. Algorithm 4.12 (TCA-construction) is on-line, which means that the TCA is built while the input tree is being read. Moreover, if at any iteration of Algorithm 4.12 a tree is indexed by the algorithm, then all its subtrees were already indexed in previous iterations. Algorithm 4.12 is incremental, which means that a TCA that indexes one tree can be extended to index another tree simply by providing the existing TCA as an input to the algorithm together with the tree that is to be indexed.

The complexity of the construction of TCA and the size of TCA for a given tree was examined. Size of TCA is linear with respect to the input: when t is a tree with n nodes and M is a TCA($\{t\}$) (a TCA that indexes tree t), then TCA M has at most $n+1$ states, $2n+1$ pushdown store symbols and the number of transition rules is $4n$. The construction of TCA for tree t takes time $2n * log_2(n+1)$ and requires working space of size at most $2n$. However, if a hash map is used for the storage of the transition function δ of the TCA, the construction time reduces to $\mathcal{O}(2n)$ and the working space stays at $2n$.

It was shown that the above-mentioned space requirements of the TCA are the maximum space requirements of TCA reached only for trees without any subtree repeats. When a tree with subtree repeats is indexed, a substantially smaller size of the TCA can be reached after running Algorithm 4.12 (TCA-construction). It was shown that the TCA can have a logarithmic size with respect to the indexed tree. This means that Algorithm 4.12 can be used for a compression of trees.

A decompression algorithm for TCA was presented. Its correctness was proved and its time and space complexities were examined. The decompression algorithm requires time $\mathcal{O}(5n)$ and space $\mathcal{O}(2n)$ for its execution. The output of the decompression algorithm is the tree that was indexed using the TCA. It has n nodes and its prefix bar notation takes $2n$ space.

The compression and decompression performance of TCA was verified experimentally. Compression by TCA reached the performance of grammar-based compression techniques [11]. It did not achieve the compression ratio of the LZ compression methods [71, 52], but has indexing capabilities that are not present in these methods. A library implementation of TCA for compression and decompression has been created [52]. The implementation verified that when a hash map is used for storing the transition function δ of the TCA, both TCA construction and TCA decompression have $\mathcal{O}(n)$ time complexities.

An algorithm for subtree matching that uses TCA was presented. It was proved that given a tree t with n nodes and a set of trees T, the algorithm for subtree matching reports all subtrees of t that match trees in T in time $\mathcal{O}(2n)$ if hashing is used.

An algorithm for finding exact repeats of subtrees in a set of trees was presented as a natural extension of Algorithm 4.12 (TCA-construction). The algorithm for finding exact repeats takes linear time with respect to the size of the input when a hash map is used for the storage of transition function δ.

The tree compression automaton has been placed into context of finite tree automata. A conversion algorithm from a TCA into a deterministic FTA that accepts the same trees has been presented. The existence of a conversion algorithm between TCA and FTA implies that finite tree automaton is suitable for subtree indexing, compression of trees and finding of exact repeats of subtrees. The latter two applications have not been explored in [18] and TCA is thus a contribution to the theory presented there.

7.2 A full and linear index of a tree for tree patterns

The finite tree automaton as a model of computation [18] is suitable for exact subtree indexing. The TCA from this doctoral thesis and the subtree pushdown automaton from arbology [50] have also been shown to be appropriate for this purpose. However, when the subtree indexing is extended to allow patterns with wildcards, as in tree pattern matching [50], all these approaches yield automata whose size can be exponential with respect to the indexed tree.

A new method of a full and linear index of a tree for tree patterns has been presented. The index consists of the compact suffix automaton, which is a standard text index structure, and a so-called subtree jump table.

Given a subject tree T with n nodes, the indexing phase is proved to take $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space. The number of distinct tree patterns which match the tree is $\mathcal{O}(2^n)$, but the index that is built during the indexing phase requires only $\mathcal{O}(n)$ space.

The searching phase reads an input tree pattern P of size m and locates all its occurrences in the tree T. For an input tree pattern P in linear prefix notation $pref(P) = P_1SP_2S \dots SP_k$, $k \geq 1$, the searching is performed in time $\mathcal{O}(m + \sum_{i=1}^{k} |occ(P_i)|))$, where $occ(P_i)$ is the set of all occurrences of string P_i in $pref(T)$.

Compared to an FTA-based index, to a TCA and to a tree pattern pushdown automaton, the searching phase has no longer strictly linear complexity, but the size of the index is linear instead of worst-case exponential.

The presented algorithms for tree pattern indexing can be extended for unranked trees when the prefix bar linear notation of the tree [50] is used instead of the prefix notation.

7.3 Space Requirements of an Index of a Tree for Tree Patterns

This doctoral thesis showed trees such that the deterministic FTAs that index them have exponential size $\mathcal{O}(2^(n/4))$ with respect to the size n of the indexed trees, whereas the non-deterministic FTAs that index them have linear size. The same result is proved for the same kind of trees for tree pattern PDAs. Examples of trees for which the same types of indexes have linear size were shown. Several properties of trees that affect the size of the investigated indexes were examined.

The linear index presented in Section 5.1 has linear size. The complexity of the searching phase is not strictly linear with respect to the size of the pattern, but depends on the structure of the indexed tree.

7.4 Suggestions for further research

7.4.1 Tree compression automaton

1. The tree compression automaton can compress a tree and provide an index of it at the same time. It can compress efficiently trees that have a recursive structure, like full k-ary trees or Fibonacci trees. However, many trees, especially in XML databases, have a structure where one parent has multiple child subtrees that have the same structure. Currently TCA does not use these repeats for further compression. TCA would achieve better compression ratio if the repeating child subtrees could be compressed, for example into a pair (subtree identifier, number of repeats).

2. The TCA could be used for indexing of a tree for tree patterns. However, a straightforward approach that extends the TCA for this purpose can lead to a rapid increase

of the number of its states. Methods for extension of TCA for tree patterns should be examined. Firstly, methods that keep the pushdown automaton as the model of computation. As these methods will probably result in a quadratic or even exponential blow-up in the number of states of the TCA, methods that use more powerful models of computation should be examined, similarly as Chapter 5 examined the index of a tree for tree patterns to overcome limits faced by tree pattern pushdown automaton [50].

3. The TCA compression library [52] should be extended to allow not only compression of trees, but to include compressed text data of tree nodes. The library should also be extended to accept XML files for input. With these extensions, the library will be applicable for compression and indexing of large XML files.

4. This whole doctoral thesis focused on ordered trees. A natural extension of the presented results is an application for unordered trees. With minor modifications, the algorithms that work with TCA could be used for unordered trees. The structure of TCA is optimized for ordered trees. It is a question whether and how its structure could be modified to allow for fast queries over unordered trees.

7.4.2 A full and linear index of a tree for tree patterns

1. To offer a linear-time searching capability, the index currently requires an $\mathcal{O}(n)$ extra space (array $Pairs$), where n is the number of nodes of the indexed tree. Although the presence of this extra space does not change the asymptotic space complexity of the indexing phase or the searching phase, it would be interesting to find an algorithm that can avoid using this extra space for the searching phase and maintain the asymptotic time complexity of the searching phase. Techniques used for linear-time construction of suffix trees or of compact suffix automaton might be useful.

2. It remains an open question whether it is possible to build a linear-size index of a tree for tree patterns that would allow purely linear-time queries in the searching phase. This doctoral thesis suggests that if such an index exists, a pushdown automaton is a computational model that might not be powerful enough for this purpose.

7.4.3 Space Requirements of an Index of a Tree for Tree Patterns

1. The exponential sizes of pushdown automaton-based indexes proved in this Doctoral thesis suggest the following proposition:

   Proposition 7.1 Let $I(t)$ denote the set of deterministic pushdown automata (DPDA) that accept all tree patterns that match a tree t. Then for any k, $k > 0$, there exists a tree t_k with n nodes, $n > k$, such that all DPDA from set $I(t_k)$ have size at least 2^{m*n}, where m is a constant shared by all k.

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall Englewood Cliffs, N.J., 1972.

[2] Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, November 2000.

[3] Arbology www pages. URL: `http://www.arbology.org/`, 2013. May 2013.

[4] Helen M. Berman, John D. Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic Acids Research*, 28(1):235–242, 2000.

[5] P. Bille. *Pattern Matching in Trees and Strings*. PhD thesis, FIT University of Copenhagen, Copenhagen, 2008.

[6] Philip Bille, Inge Li Gortz, Hjalte Wedel Vildhoj, and Soren Vind. String indexing for patterns with wildcards. In FedorV. Fomin and Petteri Kaski, editors, *Algorithm Theory - SWAT 2012*, volume 7357 of *Lecture Notes in Computer Science*, pages 283–294. Springer Berlin Heidelberg, 2012.

[7] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind. String matching with variable length gaps. In *Proceedings of the 17th international conference on String processing and information retrieval*, SPIRE'10, pages 385–394, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Anselm Blumer, J. Blumer, Andrzej Ehrenfeucht, David Haussler, and Ross M. McConnell. Linear size finite automata for the set of all subwords of a word - an outline of results. *Bulletin of the EATCS*, 21:12–20, 1983.

[9] Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985.

[10] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International*

Conference on Management of Data, SIGMOD '02, pages 310–321, New York, NY, USA, 2002. ACM.

[11] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Grammar-based tree compression. Technical report, Universitat Oldenburg, Universitt Stuttgart, 2004.

[12] Barbara Catania, Anna Maddalena, and Athena Vakali. XML document indexes: A classification. *IEEE Internet Computing*, 9(5):64–71, September 2005.

[13] Barbara Catania, Beng Chin Ooi, Wenqiang Wang, and Xiaoling Wang. Lazy XML updates: Laziness as a virtue, of update and structural join efficiency. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 515–526, New York, NY, USA, 2005. ACM.

[14] Michalis Christou, Maxime Crochemore, Tomáš Flouri, Costas S. Iliopoulos, Jan Janoušek, and Bořivoj Melichar Solon P. Pissis. Computing all subtree repeats in ordered ranked trees. In *SPIRE*, pages 338–343, 2011.

[15] Michalis Christou, Tomáš Flouri, Costas S. Iliopoulos, Jan Janoušek, Bořivoj Melichar, Solon P. Pissis, and Jan Žďárek. Tree template matching in unranked ordered trees. *J. Discret. Algorithms*, 20:51–60, May 2013.

[16] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. Apex: An adaptive path index for XML data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 121–132, New York, NY, USA, 2002. ACM.

[17] L. Cleophas. *Tree Algorithms. Two Taxonomies and a Toolkit.* PhD thesis, Technische Universiteit Eindhoven, Eindhoven, 2008.

[18] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. URL: `http://www.grappa.univ-lille3.fr/tata`, 2008. release November, 18th 2008.

[19] Maxime Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.

[20] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings.* Cambridge Univ Pr, 2007.

[21] Maxime Crochemore, Costas Iliopoulos, Christos Makris, Wojciech Rytter, Athanasios Tsakalidis, and Kostas Tsichlas. Approximate string matching with gaps. *Nordic J. of Computing*, 9(1):54–65, March 2002.

[22] Maxime Crochemore and Wojciech Rytter. *Text Algorithms.* Oxford University Press, 1994.

[23] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology.* World Scientific, 2002.

[24] Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching*, pages 116–129. Springer, 1997.

[25] Maxime Crochemore and Renaud Vérin. On compact directed acyclic word graphs. In *Structures in Logic and Computer Science*, pages 192–211. Springer, 1997.

[26] Document object model (dom) level 3 core specification. URL: `http://www.w3.org/TR/DOM-Level-3-Core/`, 2015.

[27] Andrzej Ehrenfeucht, Ross M. McConnell, Nissa Osheim, and Sung-Whan Woo. Position heaps: A simple and dynamic text indexing data structure. *J. Discrete Algorithms*, 9(1):100–121, 2011.

[28] M. J. Fischer and M. S. Paterson. String-matching and other products. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[29] T. Flouri. *Pattern Matching in Tree Structures.* PhD thesis, FIT Czech Technical University, Prague, 2013.

[30] Tomáš Flouri, Costas S. Iliopoulos, Jan Janoušek, Bořivoj Melichar, and Solon P. Pissis. Tree template matching in ranked ordered trees by pushdown automata. *J. of Discrete Algorithms*, 17:15–23, December 2012.

[31] Tomáš Flouri, Jan Janoušek, Bořivoj Melichar, Costas S. Iliopoulos, and Solon P. Pissis. Tree indexing by pushdown automata and repeats of subtrees. In *FedCSIS*, pages 899–902, 2011.

[32] Tomáš Flouri, Bořivoj Melichar, and Jan Janoušek. Subtree matching by deterministic pushdown automata. In *IMCSIT*, pages 659–666, 2009.

[33] Tomáš Flouri, Kunsoo Park, Kimon Frousios, Solon P. Pissis, Costas S. Iliopoulos, and German Tischler. Approximate string-matching with a single gap for sequence alignment. In *Proceedings of the 2Nd ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, BCB '11, pages 490–492, New York, NY, USA, 2011. ACM.

[34] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[35] F. Gecseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3 Beyond Words. Handbook of Formal Languages, pages 1–68. Springer, 1997.

[36] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[37] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.

[38] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Boston, 2nd edition, 2001.

[39] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.

[40] Jan Janoušek. *Arbology: Algorithms on Trees and Pushdown Automata. Habilitation thesis.* TU FIT, Brno, 2010.

[41] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. Xr-tree: indexing XML data for efficient structural joins. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 253–264, March 2003.

[42] Tomasz Jurdzinski and Krzysztof Lorys. Lower bound technique for length-reducing automata. *Inf. Comput.*, 205(9):1387–1412, 2007.

[43] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[44] Jan Lahoda and Jan Žďárek. Simple tree pattern matching for trees in the prefix bar notation. *Discrete Applied Mathematics*, 163, Part 3:343–351, January 2014.

[45] Moshe Lewenstein. Indexing with gaps. In Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri, editors, *String Processing and Information Retrieval*, volume 7024 of *Lecture Notes in Computer Science*, pages 135–143. Springer Berlin Heidelberg, 2011.

[46] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[47] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[48] Christian Mathis, Theo Härder, and Karsten Schmidt. Storing and indexing XML documents upside down. *Computer Science-Research and Development*, 24(1-2):51–68, 2009.

[49] Christian Mathis, Theo Härder, Karsten Schmidt, and Sebastian Bächle. XML indexing and storage: Fulfilling the wish list. *Comput. Sci.*, 30(1):51–68, February 2015.

[50] Bořivoj Melichar, Jan Janoušek, and Tomáš Flouri. Arbology: Trees and pushdown automata. *Kybernetika*, 48(3):402–428, 2012.

[51] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 277–295, London, UK, UK, 1999. Springer-Verlag.

[52] Robin Obůrka. Implementation of tree compression pushdown automaton. Bachelor thesis. ČVUT FIT, Prague, 2013. Sources available at `https://gitlab.fit.cvut.cz/arbology-group/Full-Linear-Index-For-Tree-Pattern-Matching-Implementation`.

[53] P. Mark Pettovello and Farshad Fotouhi. Mtree: An XML xpath graph index. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 474–481, New York, NY, USA, 2006. ACM.

[54] M.Sohel Rahman, CostasS. Iliopoulos, Inbok Lee, Manal Mohamed, and WilliamF. Smyth. Finding patterns with variable length gaps or dont cares. In DannyZ. Chen and D.T. Lee, editors, *Computing and Combinatorics*, volume 4112 of *Lecture Notes in Computer Science*, pages 146–155. Springer Berlin Heidelberg, 2006.

[55] Praveen Rao and Bongki Moon. PRIX: Indexing and querying XML using Prüfer sequences. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 288–299. IEEE, 2004.

[56] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.

[57] Project home page. URL: `http://www.saxproject.org/`, 2015.

[58] Jan Trávníček, Jan Janoušek, and Bořivoj Melichar. Nonlinear tree pattern pushdown automata. In *FedCSIS*, pages 871–878, 2011.

[59] Esko Ukkonen. Lower bounds on the size of deterministic parsers. *J. Comput. Syst. Sci.*, 26(2):153–170, 1983.

[60] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[61] Leonoor van der Beek, Gosse Bouma, Rob Malouf, and Gertjan van Noord. The alpino dependency treebank. In *CLIN*, pages 8–22, 2001.

[62] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. Vist: a dynamic index method for querying XML data by tree structures. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 110–121. ACM, 2003.

[63] Wei Wang, Jiang Haifeng, Hongzhi Wang, Xuemin Lin, Hongjun Lu, and Jianzhong Li. Efficient processing of XML path queries using the disk-based f&b index. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 145–156. VLDB Endowment, 2005.

[64] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[65] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[66] Cathy H. Wu, Rolf Apweiler, Amos Bairoch, Darren A. Natale, Winona C. Barker, Brigitte Boeckmann, Serenella Ferro, Elisabeth Gasteiger, Hongzhan Huang, Rodrigo Lopez, Michele Magrane, Maria Jesus Martin, Raja Mazumder, Claire O'Donovan, Nicole Redaschi, and Baris E. Suzek. The universal protein resource (uniprot): an expanding universe of protein information. *Nucleic Acids Research*, 34(Database-Issue):187–191, 2006.

[67] XML linking language (xlink) version 1.1. URL: `http://www.w3.org/TR/xlink11/`, 2015.

[68] XML path language (xpath) 3.1. URL: `http://www.w3.org/TR/xpath-31/`, 2015.

[69] XML pointer language (xpointer). URL: `http://www.w3.org/TR/xptr/`, 2015.

[70] W3c XML query (xquery). URL: `http://www.w3.org/XML/Query/`, 2015.

[71] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Publications of the Author

Reviewed Relevant Publications of the Author

[A.1] M.Poliak, J. Janoušek, B. Melichar *Tree Compression Pushdown Automaton.* Kybernetika, vol. 48, No. 3, pp. 429-452, 2012.

[A.2] M. Poliak, J. Janoušek, J. Trávníček, R. Polách, B. Melichar *A Full and Linear Index of a Tree for Tree Patterns.* 16th Descriptional Complexity of Formal Systems, Turku, Finland, *LNCS*, Vol. 8614 , pp. 198-209, 2014.

Relevant Publications of the Author Undergoing Review

[A.3] M. Poliak, J. Janoušek *On Space Requirements of Indexes of a Tree for Tree Patterns Based on Deterministic Pushdown Automata.* Submitted for publication to Theoretical Computer Science (2017), waiting for reviews.

Remaining Relevant Publications of the Author

[A.4] M. Poliak *Arbology - indexing trees with the use of pushdown automata.* Doctoral Study Report, Faculty of Information Technology, Prague, Czech Republic, 2012.

Other Publications of the Author

[A.5] J. Feyereisl, M. Nikl, M. Poliak, M. Stransky, M. Vlasak *General AI Challenge Round One : Gradual Learning.* Invited submission; to appear in: EGPAI 2017, Melbourne, Australia