FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

(Nonlinear) Tree Pattern Indexing and Backward Matching

by

Jan Trávníček

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

Dissertation degree study programme: Informatics
Department of Theoretical Computer Science

Prague, August 2018

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 and Contributions

Trees are one of the fundamental data structures used in Computer Science. The dissertation thesis contributions are best categorised as a part of arbology research [52]. Arbology research is a counterpart of stringology research. Arbology research deals with trees represented in some linear notations, i.e. like strings with additional properties that encode the tree structure. Many algorithms belonging to the stringology maybe, with some care, adapted to handle trees represented as strings using some linear notation.

This dissertation thesis is focused on finding all occurrences of tree patterns and nonlinear tree patterns inside a subject tree. Two different general approaches of solving the problem are explored in the dissertation thesis. The first approach is focused on preprocessing of the subject tree and forming a complete index of the subject tree capable of reporting the occurrences when queried with (nonlinear) tree patterns. The second approach is complementary to indexing and it is focused on preprocessing of the (nonlinear) tree pattern and creation of a matching algorithm.

The results of the dissertation thesis are divided into two parts. The first, indexing, approach is covered by two different tree indexes. The second, matching, approach is covered by a single tree pattern matching algorithm designed for various tree representations.

The first approach is represented by a *nonlinear tree pattern pushdown automaton*, which can be used to locate occurrences of (nonlinear) tree patterns and a *full and linear index* also capable of locating occurrences of tree patterns and in extended variant also of nonlinear tree patterns.

The second approach is represented by a *backward linearised tree pattern matching algorithm*, which is a variant on backward pattern matching algorithm known from the area of strings. The algorithm is designed to work with many linear representations of trees. An extension of this algorithm for nonlinear tree patterns is also presented.

Tree pattern is a representation of a subgraph of a tree, which is rooted in some node of the tree and contains a wildcard symbol in leaves representing any subtree. The nonlinear tree pattern additionally contains nonlinear variables in leaves which represent any subtree again, however, the same nonlinear variables represent the same subtrees.

Given a tree with n nodes, the number of distinct tree patterns and nonlinear tree

patterns can be at most $2^{n-1} + n - 1$ and at most $(2 + v)^{n-1} + n - 1$, respectively, where v is the number of nonlinear variables allowed in the nonlinear tree patterns.

Nonlinear tree pattern pushdown automaton

An acyclic pushdown automaton, the *nonlinear tree pattern pushdown automaton*, constructed for an ordered tree is presented. This automaton accepts all tree patterns and nonlinear tree patterns, which match the tree the automaton was constructed for. It therefore and represents a full index of the tree for such patterns.

The nonlinear tree pattern pushdown automaton is presented in three variants with decreasing space requirements, where the smallest version of the automaton has $\mathcal{O}(n^2)$ states. The automaton is nondeterministic, however, it is also so-called input–driven and therefore, it can be determinised.

A full and linear index of a tree for tree patterns

Another method of indexing a tree for (nonlinear) tree patterns is presented. Given a subject tree t with n nodes, the tree is preprocessed and an index, which consists of any standard string index structure of size $\mathcal{O}(n)$ and a *subtree jump table*, is constructed. The size of the index is then $\mathcal{O}(n)$.

The searching phase uses the index, reads a tree pattern p of size m and computes the list of positions of all occurrences of the pattern p in the tree t. 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 the longest tree pattern parts p_i in tree t, which do not contain the wildcard symbol.

The index is extended to support queries by nonlinear tree patterns when a subtree repeats table is added to the index. The size of the index and query times remain the same for the extended index.

Backward linearised tree pattern matching algorithm

A backward linearised tree pattern matching algorithm for ordered trees is presented. The algorithm finds all occurrences of a single tree pattern. The algorithm is based on backward string pattern matching algorithm that uses the bad character shift table. The algorithm preserves the properties and advantages of this backward string pattern matching algorithm. The number of symbol comparisons in the backward linearised tree pattern matching can still be sublinear with respect to the size of the input tree. As in the case of backward string pattern matching, the size of the bad character shift table used by the algorithm is linear in the size of the alphabet.

The algorithm variants for different linear tree notations are presented in details. The algorithm is modified to report occurrences of nonlinear tree patterns as well, again using the subtree repeats table.

Keywords:
Tree pattern matching, nonlinear tree pattern matching, indexing trees, pushdown automata, backward pattern matching, tree processing, tree linearisation

Acknowledgements

First of all, I would like to express my gratitude to my dissertation thesis supervisor, Jan Janoušek, Ph.D., associate professor of computer science. He has been guiding me not only during the Ph.D. studies but also during the MSc. studies when he had given me the opportunity to continue my academic studies. He helped me especially in the first years when the support was most needed. I also thank him for many hours of consultations and scientific discussions that pushed me further in my professional career.

A special thanks goes to professor Bořivoj Melichar for co-tutoring me during my Ph.D. studies in the area of stringology and arbology. His constant encouragement to simplify every single definition and algorithm has been a source of greater understanding that would otherwise remain unseen. I also thank for his patient approach in revising my first publications.

Next, great thanks go to the staff of the Department of Theoretical Computer Science, namely professor Jan Holub, Ph.D., Ladislav Vagner, Ph.D., and Jan Žďárek, Ph.D. for additional support in my teaching career. To my Ph.D. colleagues and recently finished Ph.D. graduates, namely Tomáš Pecka, Radomír Polách, Martin Šlapák, Ondřej Guth, Martin Poliak, Eliška Šestáková, Petr Máj, and many others for having been a source of a pleasant work environment. To all members of Prague Stringology Club for additional support in the research career.

Many thanks go to Jan Janoušek, Bořivoj Melichar, Jan Holub, and Jakub Jaroš who attracted me to the field of stringology, arbology, and automata theory and essentially gave me an idea to apply for Ph.D. study.

I would like to express special thanks to the department, faculty, and university management for providing most of the funding for my research.

Another thanks belong to Loek Cleophas for co-authoring a scientific publication, valuable comments, and beneficial cooperation.

Last but not least, my greatest thanks go to my family members. To my wife for being a constant source of support and for infinite patience. To my parents for their care and support in my studies.

My research has been partially supported by the Czech Science Foundation (GAČR) as project No. GA-13-03253S and by the Grant Agency of the Czech Technical University in Prague, grant No. SGS17/209/OHK3/3T/18.

Dedication

To my wife Marťa, my mom, and my dad

Contents

List of Figures

List of Tables

List of Algorithms

*

List of Trees and Tree Patterns

Trees

Tree t_{1r}	$pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$
Tree t_{1u}	$pref\_bar(t_{1u}) = a\ a\ a\ \uparrow\ a\ a\ \uparrow\uparrow\uparrow\ a\ a\ \uparrow\uparrow\uparrow$
Tree t_{2r}	$pref(t_{2r}) = a4\ a4\ a4\ a0\ b0\ a0\ a0\ a0\ b0\ a0\ a0\ a0\ b0$
Tree t_{3r}	$pref(t_{3r}) = a(n-1)\ a0\ a0 \ldots a0$, where $n \geq 2$
Tree t_{4r}	$pref(t_{4r}) = a2\ a2\ a0\ a0a2\ a0\ a0$
	$post\ (t_{4r}) = a0\ a0\ a2\ a0\ a0\ a2\ a2$
	$pref\_ranked\_bar\ (t_{4r}) = a2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ \uparrow 2$
	$post\_ranked\_bar\ (t_{4r}) = \uparrow 2\ \uparrow 2\ \uparrow 0\ a0\ \uparrow 0\ a0\ a2\ \uparrow 2\ \uparrow 0\ a0\ \uparrow 0\ a0\ a2\ a2$
Tree t_{5r}	$pref\_ranked\_bar\ (t_{5r}) = a2\ a2\ b0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ \uparrow 2$

Tree Patterns

Tree pattern p_{1r}	$pref(p_{1r}) = a2\ a0\ a1\ a0$
Tree pattern p_{2r}	$pref(p_{2r}) = a2\ S\ a1\ S$
Tree pattern p_{3r}	$pref(p_{3r}) = a2\ X\ a1\ X$
Tree pattern p_{4r}	$pref(p_{4r}) = a4\ b0\ a0\ a0\ a0$
Tree pattern p_{5r}	$pref(p_{5r}) = a4\ S\ a0\ S\ S$
Tree pattern p_{6r}	$pref(p_{6r}) = a4\ S\ a0\ X\ X$
Tree pattern p_{7r}	$pref(p_{7r}) = a4\ Z\ Z\ Y\ Y$
Tree pattern p_{8r}	$pref(p_{8r}) = a2\ a1\ S\ a1\ a0$
	$post(p_{8r}) = S\ a1\ a0\ a1\ a2$
	$pref\_ranked\_bar(p_{8r}) = a2\ a1\ S\ \uparrow S\ \uparrow 1\ a1\ a0\ \uparrow 0\ \uparrow 1\ \uparrow 2$
	$post\_ranked\_bar(p_{8r}) = \uparrow 2\ \uparrow 1\ \uparrow S\ S\ a1\ \uparrow 1\ \uparrow 0\ a0\ a1\ a2$
Tree pattern p_{9r}	$pref\_ranked\_bar(p_{9r}) = a1\ S\ \uparrow S\ \uparrow 1$
Tree pattern p_{10r}	$pref(p_{10r}) = a2\ S\ S$
	$post(p_{10r}) = S\ S\ a2$
	$pref\_ranked\_bar(p_{10r}) = a2\ S\ \uparrow S\ S\ \uparrow S\ \uparrow 2$
	$post\_ranked\_bar(p_{10r}) = \uparrow 2\ \uparrow S\ S\ \uparrow S\ S\ a2$
Tree pattern p_{11r}	$pref\_ranked\_bar(p_{11r}) = a2\ a1\ a0\ \uparrow 0\ \uparrow 1\ a1\ S\ \uparrow S\ \uparrow 1\ \uparrow 2$
Tree pattern p_{12r}	$pref\_ranked\_bar(p_{12r}) = a2\ X\ \uparrow X\ X\ \uparrow X\ \uparrow 2$

Abbreviations

Alphabet, Language, String

$\mathcal{A}$	Alphabet
L	Language
ε	Empty string
$\mathcal{A}^*$	Set of all strings over $\mathcal{A}$, including ε
$\mathcal{A}^+$	$\mathcal{A}^* \setminus \{\varepsilon\}$
$\lvert A \rvert$	Number of elements in set A

Graph, Tree

G	Graph
N	Set of nodes
R	Set of edges
t	Tree
p	(Nonlinear) Tree pattern
r	Root node
$pref$	Prefix notation
$post$	Postfix notation
$pref\_bar$	Prefix bar notation
$post\_bar$	Postfix bar notation
$pref\_ranked\_bar$	Prefix ranked bar notation
$post\_ranked\_bar$	Postfix ranked bar notation
S	Placeholder for any subtree
$X, Y, \ldots$	Nonlinear variables

Finite Automaton

FA	Finite automaton
DFA	Deterministic Finite Automaton
NFA	Nondeterministic Finite Automaton
Q	Set of states
q_0	initial state
F	final states

Finite Tree Automaton

FTA	Finite tree automaton
$DFRTA$	Deterministic Frontier to Root Tree Automaton
$DRFTA$	Deterministic Root to Frontier Tree Automaton
$NFRTA$	Nondeterministic Frontier to Root Tree Automaton
$NRFTA$	Nondeterministic Root to Frontier Tree Automaton
Q	Set of states
F	Set of final states

Pushdown Automaton

PDA, M	Pushdown automaton
Q	Set of states
G	Pushdown store alphabet
q_0	Initial state
Z_0	Elements of pushdown store alphabet

Main Results in Tree Indexing, Basic notions, Previous Results and Related Work

FA_b	Backbone of the nondeterministic suffix automaton
FA_{nsuf}	Nondeterministic suffix automaton
FA_{csuf}	Compact suffix automaton
M_p	Backbone of the nondeterministic subtree pushdown automaton
M_{pt}	Tree top pushdown automaton
M_{tnpt}	Tail of nondeterministic tree pattern pushdown automaton
M_{nps}	Nondeterministic subtree pushdown automaton
M_{dps}	Deterministic subtree pushdown automaton
M_{npt}	Nondeterministic tree pattern pushdown automaton
M_{dpt}	Deterministic tree pattern pushdown automaton
$srms$	Set of subtree rightmost states
SJT	Subtree jump table
TPP	Tree pattern prefix
PH	Position Heap
$DAWG$	Directed Acyclic Word Graph
ms	Mergeable States
mss	Mergeables States Set
sms	Simple Mergeable States
$smss$	Simple Mergeable States Set
nnv	Number of Nonlinear Variables
$tnst$	Tree Node State Label
tdn	Tail Denoting State
M_{dnpt}	Deterministic nonlinear tree pattern automaton
M_{nnpt}	Nondeterministic nonlinear tree pattern automaton
M_b	Basic nonlinear tree pattern automaton
M_{db}	Basic deterministic nonlinear tree pattern automaton
M_s	Simple nonlinear tree pattern automaton
M_{ds}	Simple deterministic nonlinear tree pattern automaton

Main Results in Tree Pattern Matching, Basic notions, Previous Results and Related Work

BCS	Bad Character Shift
GSS	Good Suffix Shift
MS	Match Shift
LTPM	Linearised Tree Pattern Matching
RBCS	Reversed Bad Character Shift
SJT	Subtree jump table

Other

GNU	GNU's Not Unix
GCC	GNU Compiler Collection
XML	eXtended Markup Language

Chapter 1

Introduction

1.1 Motivation

Trees are one of the fundamental data structures used in Computer Science and the theory of formal tree languages has been extensively studied and developed since the 1960s [17, 33]. Tree pattern matching on node-labelled trees is an important algorithmic problem with applications in many tasks such as compiler code selection, interpretation of nonprocedural languages, implementation of rewriting systems, and processing XML or markup languages in general. Tree patterns are trees whose leaves can be labelled by a special wildcard, the wildcard symbol S, which serves as a placeholder for any subtree.

Since the linear notation of a subtree of a tree is a substring of the linear notation of that tree, the subtree matching and tree pattern matching problems are in many ways similar to the string pattern matching problem. However, linear notations of trees are generated by context-free grammars while strings are generated by regular grammars. Furthermore, we note that the tree pattern matching problem is more complex than the string matching one because there can be at most n^2 distinct substrings of a string of size n, whereas there can be at most $2^{n-1} + n$ distinct tree patterns which match a tree of size n.

Nonlinear tree patterns can further contain leaves labelled by specific nonlinear variable symbols X, Y, $\ldots$, where each of these symbols represents a specific subtree. Given a tree with n nodes, the numbers of distinct tree patterns and nonlinear tree patterns which match the tree can be at most $2^{n-1} + n - 1$ and at most $(2 + v)^{n-1} + n - 1$, respectively, where v is the maximal number of distinct nonlinear variables allowed in nonlinear tree patterns.

1.2 Problem statement

The (nonlinear) tree pattern matching problem is to locate occurrences of (nonlinear) tree pattern p of size m in subject tree t of size n, i.e. positions of nodes of subject tree t rooting a subtree of subject tree t which match (nonlinear) tree pattern p. The (nonlinear) tree pattern can be either a tree itself, tree pattern, or nonlinear tree pattern and the problem is referred to as subtree matching, tree pattern matching and nonlinear tree pattern matching, respectively.

The first approach can be by preprocessing the subject tree resulting in an index that can be queried with (nonlinear) tree patterns. The second approach can be by preprocessing the (nonlinear) tree pattern resulting in a matcher that can be run on the subject tree.

An efficient algorithm of indexing trees for (nonlinear) tree pattern matching and an efficient algorithm for (nonlinear) tree pattern matching is requested. The algorithms shall be defined with a focus on simplicity on design either by a simple extension of existing algorithms on strings or by designing the algorithm formally with use of automata.

1.3 Related work/previous results

The theory of formal tree languages have been extensively studied and developed since the 1960s and its main models of computation are various kinds of tree automata [33, 13, 17]. As mentioned, trees can be linearised into strings. Such a linear notation can be obtained by a corresponding tree traversal. Moreover, every sequential algorithm on a tree traverses its nodes in a sequential order, which corresponds to some linear notation. Such a linear representation need not be built explicitly. It is proved that the deterministic pushdown automaton (PDA) is an appropriate model of computation for labelled ordered trees in linear notation [43] and that the trees in postfix notation acceptable by deterministic PDA form a proper superclass of the class of regular tree languages [33], which are accepted by finite tree automata. Recently, pushdown automata gain popularity in solving practical problems of processing trees, for example in processing XML documents [32].

The theory of text indexing, which is a result of stringology research [19, 21, 22, 23, 51, 58], is very well-researched and uses many sophisticated data structures: suffix tree and suffix array are most widely used structures for text indexing, providing efficient solutions for a wide range of applications. The Directed Acyclic Word Graph [6], also known as suffix (or factor) automaton [18], is another elegant structure. Compact suffix automaton represents the minimised and compacted version of suffix trees and suffix automata, respectively [24]. Another text indexing structure, called position heap, was proposed recently [27]. 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. By means of the suffix tree, the compact suffix automaton, or the position heap, the list of positions of all occurrences of an input string pattern y of size m can be computed in time $\mathcal{O}(m + |occ(y)|)$, where $occ(y)$ is the set of all occurrences of y in the text [21].

A subtree pushdown automaton [41] which represents a complete index of an ordered tree for subtrees is an example of an indexing structure in an area of trees.

The problem of matching string patterns with gaps has been explored in many methods. The methods differ in the kinds of considered gaps, in the achieved complexity and in the fact whether the method is based on indexing, where the subject text is preprocessed, or it is based on the principle that a string pattern is preprocessed and the subject text is read as the input of the searching phase. A method of indexing a text for string patterns with gaps is described in [4], where an index is constructed for matching with wildcards. A wildcard matches any single character, which cannot be used for tree patterns in a linear notation. In [49], an index is constructed for matching with variable length gaps. Unfortunately, the searching time depends on the gaps sizes, which is not time efficient for matching tree patterns, where the gaps can be of any size.

Many algorithms have been proposed for exact string matching [9, 23, 28, 58]. Among the most efficient of them are those based on backward string pattern matching, represented by the Boyer-Moore and Boyer-Moore-Horspool algorithms. Although backward string pattern matching's time complexity is generally $O(n * m)$ (for text and pattern size n and m respectively) in the worst case, due to such algorithms' ability to skip text parts, they often perform sublinearly in practice.

Many tree pattern matching algorithms exist as well [10, 13, 31, 36, 57] and many of them use some kind of tree automata [13]. Cole et al. [15] use a subset matching approach, but at the cost of large auxiliary data structures. For unrestricted tree pattern sets, among the fastest algorithms in practice are algorithms based on deterministic frontier-to-root (bottom-up) tree automata (DFRTAs) [10, 13, 36] and Hoffmann-O'Donnell-style stringpath matchers [2, 36]. A few of these tree pattern matching algorithms use principles of matching patterns backwards: Hoffmann and O'Donnell refer to work by Lang et al. [48] that applies such an approach to leftmost stringpaths of trees which involves complications when dealing with nodes of arity greater than 2. Another tree pattern matching algorithm was introduced in [63], where symbols of a tree pattern and string paths of an input subject tree are compared. The shifting is based on extension of Boyer-Moore style of shifting for more patterns introduced in [16]. This algorithm can skip nodes of the subject tree when it is known that no occurrence of the pattern is skipped.

There exist two basic approaches to pattern matching problems. The first basic approach is represented by the use of an indexing data structure constructed for the subject in which the search can be implemented. In other words, the subject is preprocessed. Given a tree pattern of size m, such an index can locate the tree pattern typically in time linear in m. This approach is suitable especially for cases when multiple queries to find occurrences of different input patterns in a given subject structure are performed. The second approach is represented by the use of a pattern matcher which is constructed for patterns. In other words, the patterns are preprocessed. Given a tree of size n, such tree pattern matcher perform the search phase typically in time linear in n. This approach is suitable for cases when the task is to look for occurrences of a given pattern in many subject trees.

1.4 Contributions of the thesis

The thesis presents four following contributions:

1. The modification of tree pattern matching automata for nonlinear tree pattern matching.

2. A new efficient method of simulation of the nonlinear tree pattern pushdown automata as the next step in indexing of trees for nonlinear pattern matching.

3. A new and first algorithm of backward linearised tree pattern matching algorithm for tree pattern matching.

4. The modification of backward tree pattern matching algorithm for nonlinear tree pattern matching.

Representatives of the first approach are acyclic pushdown automata for an ordered tree and a combination of compact suffix automaton with additional structure to efficiently handle wildcard symbols in tree patterns.

The *tree pattern pushdown automaton* and the *nonlinear tree pattern pushdown automaton* represent a complete index of the tree for tree patterns and nonlinear tree patterns, respectively, and accept all tree patterns and nonlinear tree patterns, respectively, which match the tree. The construction of nondeterministic (nonlinear) tree pattern pushdown automata is shown and a discussion of their time and space complexities follow. The presented nondeterministic pushdown automata are input–driven and therefore can be determinised.

We note that the presented PDAs have only one pushdown symbol and therefore can be easily transformed to counter automata, which are a weaker and simpler model of computation than the PDA. We present the automata in this paper as PDAs because the PDA is a more fundamental and more widely-used model of computation than the counter automaton.

Since our pushdown automata accept finite languages, which correspond to finite sets of connected subgraphs of the tree, a finite automaton could also be used instead of a pushdown automaton. However, such a finite automaton would have significantly more states than the PDA, in which the pushdown store efficiently processes the underlying tree structure.

Tree pattern pushdown automaton represents a full index of a tree for tree patterns but its size is not linear to the size of the subject tree [52]. Also, a finite tree automaton accepting all tree patterns that match the tree can be constructed but its size is exponential to the size of the subject tree [13, 17, 42].

In [5], a preprocessing of a pattern and subsequently a matching algorithm for variable length gap matching problem was proposed. This solution is incompatible with the tree pattern matching problem because of different interpretation of gaps. Moreover, this solution is not indexing but matching, when the pattern is preprocessed. By analogy for trees, many tree pattern matching methods, which preprocess tree patterns and not the subject tree, have been proposed [13, 36].

Another new and simple method of indexing a tree for tree patterns is presented as a continuation of proposed (nonlinear) tree pattern pushdown automata. Given a subject tree t with n nodes, the tree is preprocessed and an index, which consists of a standard string compact suffix automaton and a subtree jump table, is constructed. We note that any convenient text index can be used instead of the compact suffix automaton, which has been chosen here because of its good space and time complexities. Despite the fact that the number of distinct tree patterns which match the tree is $\mathcal{O}(2^n)$, the size of the index presented in this paper is $\mathcal{O}(n)$.

The searching phase uses the index, reads a tree pattern p of size m and computes the list of positions of all occurrences of the pattern p in the tree t. For a 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 p_i in $pref(t)$. We are not aware of any other known method of full and linear indexing a tree for tree patterns with these time and space complexities.

A representative of the second approach (matching) is an algorithm that uses a linear representation of the subject tree where random access to symbols/positions is possible and it is based on Boyer-Moore-Horspool's algorithm. While modifying backward string pattern matching to backward subtree matching (searching for occurrences of given subtrees) is straightforward, this is not the case for backward tree pattern matching, where complications arise due to the use of wildcard symbol S and matched subtrees being possibly recursively nested. A new backward tree pattern matching algorithm is presented. The presented backward tree pattern matching algorithm preserves the properties and the advantages of the standard backward string pattern matching: the number of symbol comparisons in the backward tree pattern matching can be sublinear in n, the size of the subject tree. Based on the Boyer-Moore-Horspool algorithm, a modified bad character shift heuristic is used. As in the case of backward string pattern matching, the size of the bad character shift table used by the algorithm is linear with the size of the alphabet. Our experimental results confirm the properties of the algorithm and show that it outperforms the DFRTAs and stringpath matchers mentioned above.

1.5 Structure of the thesis

The thesis is organised into six chapters as follows:

1. *Introduction*: Describes the motivation behind our efforts together with our goals. There is also a list of contributions of this dissertation thesis.
2. *Theoretical Background*: Introduces the reader to the necessary theoretical background of strings, trees, automata, etc.
3. *Previous Results and Related Work*: Surveys the state-of-the-art the dissertation thesisbuilds upon.
4. *Main Results in Tree Indexing*: Presents the results of the author in the area of indexing trees. Two new indexes of trees are proposed and defined.
5. *Main Results in Tree Pattern Matching*: Presents the results of the author in the area of tree pattern matching. One new tree pattern matching algorithm, including its variants, is proposed and defined.

6. *Conclusions*: Summarises the results of our research, suggests possible topics for further research and concludes the thesis.

CHAPTER 2

Theoretical Background

Basic notions including strings and trees, definitions of finite, finite tree, and pushdown automata are given in this chapter. Example trees and tree patterns, including their linearisations, used throughout the thesis are presented here as well.

This chapter also presents string indexing structures used as a backend for introduced tree indexing structure.

2.1 Alphabet, string

An *alphabet* is a finite nonempty set of *symbols*. 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 $\mathit{Arity}(a)$. The set of symbols of arity p is denoted by $\mathcal{A}_p$. Elements of arity $0, 1, 2, \ldots, p$ are called nullary (constants), unary, binary, ..., p-ary symbols, respectively. We assume that alphabet $\mathcal{A}$ contains at least one constant. In the examples, we use numbers at the end of 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 $\mathcal{A}$.

A set $\mathcal{A}_p$ where $p \geq 0$ is a ranked alphabet of symbols of arity p only.

A *string* x is a sequence of i symbols $s_1 s_2 s_3 \ldots s_i$ from a given alphabet, where i is the size of the string. A sequence of zero symbols is called the empty string. The empty string is denoted by symbol ε.

2.2 Tree, tree pattern, linear notations and their properties

Based on concepts and notations from graph theory [3]:

A *graph* G is a pair (N, R), where N is a set of nodes and R is a set of edges such that each element of R is of the form (f, g), where $f, g \in N$. This element will indicate that, for node f, there is an edge between node f and node g.

A *directed graph* G is a graph, where each element of R of the form (f, g) indicates that, there is an edge leaving node f and entering node g. This edge is ordered from f to g. An *undirected graph* G is a graph in which no such ordering of edges is given.

A sequence of nodes f_0, f_1, ..., 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 *labelling* of an ordered graph $G = (N, R)$ is a mapping of N into a set of labels. In the examples, we use a_f for a short declaration of node f labelled by symbol a.

A directed graph is *connected* if there exists a path from f_u to f_v for each pair of nodes (f_u, f_v), $u \neq v$, of the graph.

A *cycle* is a path f_0, f_1, ..., f_n in which $f_0 = f_n$.

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

A *tree* is a connected directed graph without any cycle. The tree is assumed to have at least one node. A *rooted tree* t is a tree with a special node $r \in N$, called the *root*.

The rooted tree t can also be defined by following:

(1) $r \in N$ (root) has the in-degree 0,
(2) all other nodes of t have the in-degree 1,
(3) there is just one path from the root r to every node $f \in N$, where $f \neq r$.

Nodes of a tree with the out-degree 0 are called *leaves*.

A *labelled and rooted tree* is a tree with the additional property: (4) every node $f \in N$ is labelled by a symbol $a \in \mathcal{A}$, where $\mathcal{A}$ is an alphabet.

A node g is a *direct descendant* of node f if a pair $(f, g) \in R$.

An *ordered, labelled and rooted tree* is a labelled and rooted tree where direct descendants of a node f are ordered.

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

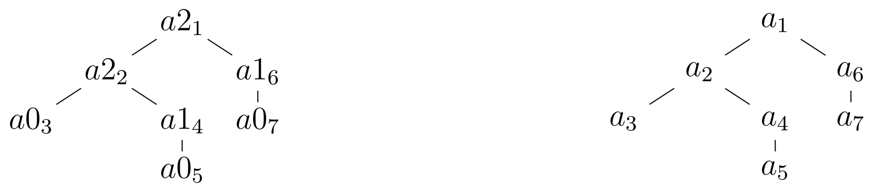
Throughout the text, a shorthand *ranked tree* and *unranked tree* will be used in the context of an ordered, ranked, labelled, and rooted tree and an ordered, labelled, and rooted tree, respectively.

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

1. $pref(a) = a0$ if a is a leaf,
2. $pref(t) = an\ pref(b_1)\ \ldots pref(b_n)$, where a is the root of tree t, $n = Arity(a)$ and $b_1, \ldots\ b_n$ are direct descendants of a.

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

1. $pref\_bar(a) =\ a \uparrow$ if a is a leaf,



(a) Tree t_{1r} from Example 2.2.1. (b) Tree t_{1u} from Example 2.2.2.

Figure 2.1: Tree t_{1r} over a ranked alphabet (left), and the same tree t_{1u} over an unranked alphabet (right) from Examples 2.2.1 and 2.2.2.

2. $pref\_bar(t) = \ a\ pref\_bar(b_1)\ \ldots pref\_bar(b_n) \uparrow$, where a is the root of tree t and $b_1, \ldots\ b_n$ are direct descendants of a.

The *postfix notation* $post(t)$ of a ranked tree t is defined as follows:

1. $post(a) = a0$ if a is a leaf,
2. $post(t) = post(b_1)\ \ldots post(b_n)\ an$, where a is the root of tree t, $n = Arity(a)$ and $b_1, \ldots\ b_n$ are direct descendants of a.

The *postfix bar notation* $post\_bar(t)$ of a unranked tree t is defined as follows:

1. $post\_bar(a) = \ a \uparrow$ if a is a leaf,
2. $post\_bar(t) = \ a\ post\_bar(b_1)\ \ldots post\_bar(b_n) \uparrow$, where a is the root of tree t and $b_1, \ldots\ b_n$ are direct descendants of a.

Example 2.2.1. Consider a ranked alphabet $\mathcal{A} = \{a2,\ a1,\ a0\}$. Consider an ordered, ranked, labelled and rooted tree $t_{1r} = (\{a2_1,\ a2_2,\ a0_3,\ a1_4,\ a0_5,\ a1_6,\ a0_7\},\ R_{t1r})$ over alphabet $\mathcal{A}$, where $R_{t1r} = \{(a2_1, a2_2), (a2_1, a1_6), (a2_2, a0_3), (a2_2, a1_4), (a1_4,\ a0_5), (a1_6, a0_7)\}$. Tree t_{1r} in the prefix notation is $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$. Trees can be represented graphically, as is done for tree t_{1r} in Figure 2.1a. △

Example 2.2.2. Consider an unranked alphabet $\mathcal{A} = \{a\}$. Consider an ordered, labelled and rooted tree $t_{1u} = (\{a_1,\ a_2,\ a_3,\ a_4,\ a_5,\ a_6,\ a_7\},\ R_{t1u})$ over an alphabet $\mathcal{A}$, where $R_{t1u} = \{(a_1, a_2), (a_1, a_6), (a_2, a_3), (a_2, a_4), (a_4, a_5), (a_6, a_7)\}$. Tree t_{1u} in the prefix bar notation is $pref\_bar(t_{1u}) = a\ a\ a\ \uparrow\ a\ a\ \uparrow\uparrow\uparrow\ a\ a\ \uparrow\uparrow\uparrow$. The tree t_{1u} is illustrated in Figure 2.1b. △

Example 2.2.3. Consider tree t_{1r} from Example 2.2.1 and its prefix, postfix, prefix bar, and postfix bar notations. The linear notations of trees are all together illustrated in Table 2.1. △

The symbols of the prefix and postfix notations of a tree correctly interleaved together make up the prefix bar notation. Symbols of the prefix notation are in essence symbols of the prefix bar notation and symbol of the postfix notation are mapped to bar symbols of the prefix bar notation. The relation of the prefix and postfix notations to the postfix bar notation is similar, with the exactly opposite correspondence of symbols.

Table 2.1: Prefix, postfix, prefix bar, and postfix bar linear notations of subject tree t_{1r}.

prefix_bar	a	a	a	$\uparrow$	a	a	$\uparrow$	$\uparrow$	$\uparrow$	a	a	$\uparrow$	$\uparrow$	$\uparrow$
postfix_bar	$\uparrow$	$\uparrow$	$\uparrow$	a	$\uparrow$	$\uparrow$	a	a	a	$\uparrow$	$\uparrow$	a	a	a
prefix	$a2$	$a2$	$a0$		$a1$	$a0$				$a1$	$a0$			
postfix				$a0$			$a0$	$a1$	$a2$			$a0$	$a1$	$a2$

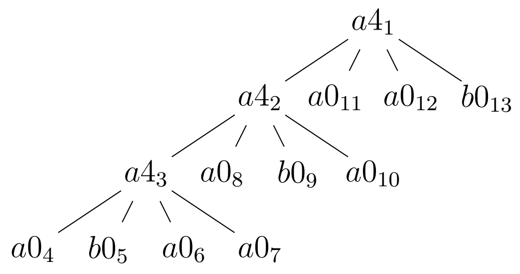


Figure 2.2: Tree t_{2r} from Example 2.2.4.

Example 2.2.4. As another example; Consider a ranked alphabet $\mathcal{A} = \{a4, a0, b0\}$. Consider an ordered, ranked, labelled, rooted, and directed tree $t_{2r} = (\{a4_1, a4_2, a4_3, a0_4, b0_5, a0_6, a0_7, a0_8, b0_9, a0_{10}, a0_{11}, a0_{12}, b0_{13}\}, R_{t2r})$ over an alphabet $\mathcal{A}$, where R_{t2r} is a set of the following ordered pairs:

$$R_{t2r} = \{(a4_1, a4_2), (a4_1, a0_{11}), (a4_1, a0_{12}), (a4_1, b0_{13}), (a4_2, a4_3), (a4_2, a0_8), \\ (a4_2, b0_9), (a4_2, a0_{10}), (a4_3, a0_4), (a4_3, b0_5), (a4_3, a0_6), (a4_3, a0_7)\}.$$

The prefix notation of tree t_{2r} is $pref(t_{2r}) = a4\ a4\ a4\ a0\ b0\ a0\ a0\ a0\ b0\ a0\ a0\ a0\ b0$. Tree t_{2r} is illustrated in Figure 2.2. $\triangle$

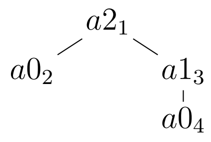
The height of tree t, denoted by *Height(t)*, is defined as the length of the longest path leading from the root of t to a leaf of t.

A *subtree* (a complete subtree) of tree $t = (N, R)$ is any tree $t' = (N', R')$ such that:

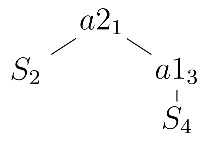
1. N' is an nonempty subset of N,
2. $R' = (N' \times N') \cap R$, and
3. No node of $N \setminus N'$ is a descendant of a node in N'.

To define a *tree pattern*, we use a special wildcard symbol $S \notin \mathcal{A}$, $Arity(S) = 0$, which serves as a placeholder for any subtree. A tree pattern is defined as an ordered, ranked, labelled and rooted tree over an alphabet $\mathcal{A} \cup \{S\}$. We will assume that the tree pattern contains at least one node labelled by a symbol from $\mathcal{A}$. A tree pattern containing at least one symbol S will be called a *tree template*.

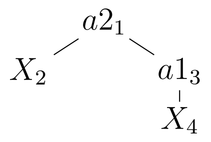
A tree pattern p with $k \geq 0$ occurrences of the symbol S *matches* a subject tree t at node n if there exist subtrees $t_1, t_2, \ldots, t_k$ (not necessarily the same) of t such that the tree p', obtained from p by substituting the subtree t_i for the i-th occurrence of S in p, $1 \leq i \leq k$,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



(a) Subtree p_{1r} from Example 2.2.5.



(b) Tree pattern p_{2r} from Example 2.2.5.



(c) Nonlinear tree pattern p_{3r} from Example 2.2.5.

Figure 2.3: Subtree, tree pattern, and nonlinear tree pattern over ranked alphabet from Example 2.2.5.

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 $pref(t) = a_1\ a_2 \ldots a_i\ a_{i+1} \ldots a_{i+m-1}\ a_{i+m} \ldots$. 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 substring $pref(t_s)$ in string $pref(t)$.

The *nonlinear tree pattern* also uses another special wildcard symbols $X, Y, \ldots$, not in alphabet $\mathcal{A}$, called nonlinear variables. These symbols serve as placeholders for specific subtrees. Every occurrence of a symbol X, Y, ... in a nonlinear tree pattern is matched with the same subtree. A nonlinear tree pattern has to contain at least one symbol from $\mathcal{A}$. A nonlinear tree pattern which contains at least two equal nonlinear variables will be called a *nonlinear tree template.*

A nonlinear tree pattern *np* with $k \geq 0$ occurrences of a the symbol S and $l_x \geq 0, l_y \geq 0, \ldots$ occurrences of nonlinear variable $X, Y, \ldots$ *matches* a subject tree t at node n if there exist subtrees $t_1, t_2, \ldots, t_k$ (not necessarily the same) of the tree t and a subtree $t_X, t_Y, \ldots$ of the tree t such that the tree *np*$'$, obtained from *np* by substituting the subtree t_i for the i-th occurrence of S in p, $1 \leq i \leq k$ and by substituting the subtree $t_X, t_Y, \ldots$ for the i-th, $1 \leq i \leq l_x, l_y, \ldots$ occurrences of $X, Y, \ldots$ in *np*, is equal to the subtree of t rooted at n.

Let $\mathcal{X}$ denote a set of nonlinear variables in the nonlinear tree pattern.

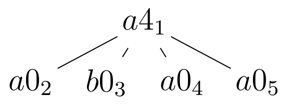
Example 2.2.5. Consider a ranked tree $t_{1r} = (\{a2_1,\ a2_2,\ a0_3,\ a1_4,\ a0_5,\ a1_6,\ a0_7\},\ R_{1r})$ from Example 2.2.1, which is illustrated in Figure 2.1a.

Consider a subtree p_{1r} over alphabet $\mathcal{A}, p_{1r} = (\{a2_1,\ a0_2,\ a1_3,\ a0_4\}, R_{p1r})$. Subtree p_{1r} in the prefix notation is $pref(p_{1r}) = a2\ a0\ a1\ a0$ and $R_{p1r} = \{((a2_1, a0_2),\ (a2_1, a1_3)),\ ((a1_3, a0_4))\}$.

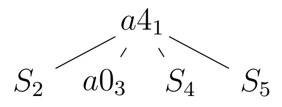
Consider a tree pattern p_{2r} over alphabet $\mathcal{A} \cup \{S\}$, $p_{2r} = (\{a2_1,\ S_2,\ a1_3,\ S_4\},\ R_{p2r})$. Tree pattern p_{2r} in the prefix notation is $pref(p_{2r}) = a2\ S\ a1\ S$ and $R_{p2r} = \{(a2_1, S_2),\ (a2_1, a1_3),\ (a1_3, S_4)\}$.

Consider a nonlinear tree pattern p_{3r} over alphabet $\mathcal{A} \cup \{S, X\}$, $p_{3r} = (\{a2_1,\ X_2,\ a1_3,\ X_4\},\ R_{p3r})$. Nonlinear tree pattern p_{3r} in the prefix notation is $pref(p_{3r}) = a2\ X\ a1\ X$ and $R_{p2r} = \{(a2_1, X_2), (a2_1, a1_3), (a1_3, X_4)\}$.

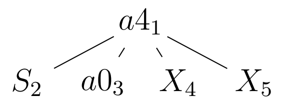
Tree patterns p_{1r}, p_{2r} and p_{3r} are illustrated in Figure 2.3. Tree pattern p_{1r} occurs once in tree t_{1r} — it matches at node 2 of t_{1r}. Tree pattern p_{2r} occurs twice in t_{1r} — it matches at nodes 1 and 2 of t_{1r}. Tree pattern p_{3r} occurs once in t_{1r} — it matches at nodes 2 of t_{1r}. △



(a) Subtree p_{4r} from Example 2.2.6.



(b) Tree pattern p_{5r} from Example 2.2.6.



(c) Nonlinear tree pattern p_{6r} from Example 2.2.6.

Figure 2.4: Subtree p_{4r} (left), tree pattern p_{5r} (center), and nonlinear tree pattern p_{6r} (right) from Example 2.2.6.

Example 2.2.6. As another example, consider tree t_{2r} from Example 2.2.4.

Consider subtree p_{4r} over alphabet $\mathcal{A}, p_{4r} = (\{a4_1, a0_2, b0_3, a0_4, a0_5\}, R_{p4r})$, $pref(p_{4r}) = a4\ b0\ a0\ a0\ a0$ and $R_{p4r} = \{(a4_1, a0_2), (a4_1, b0_3), (a4_1, a0_4), ((a4_1, b0_5)\}$.

Consider tree pattern p_{5r} over an alphabet $\mathcal{A} \cup \{S\}$, $p_{5r} = (\{a4_1,\ S_2,\ a0_3,\ S_4,\ S_5\}, R_{p5r})$. Tree pattern p_{5r} in the prefix notation is $pref(p_{5r}) = a4\ S\ a0\ S\ S$ and $R_{p5r} = \{(a4_1, S_2), (a4_1, a0_3), (a4_1, S_4), (a4_1, S_5)\}$.

Consider a nonlinear tree pattern p_{6r} over an alphabet $\mathcal{A} \cup \{S, X\}$, $p_{6r} = (\{a4_1,\ S_2,\ a0_3,\ X_4,\ X_5\},\ R_{p6r})$. Tree pattern p_{6r} in the prefix notation is $pref(p_{6r}) = a4\ S\ a0\ X\ X$ and $R_{p6r} = \{(a4_1, S_2), (a4_1, a0_3), (a4_1, X_4), (a4_1, X_5)\}$.

Tree patterns p_{4r}, p_{5r}, and p_{6r} are illustrated in Figure 2.4. Tree pattern p_{4r} has one occurrence in tree t_{2r} – it matches t_{2r} at node 3. Tree pattern p_{5r} has two occurrences in tree t_{2r} – it matches t_{2r} at nodes 1 and 2. Tree pattern p_{6r} has no occurrence in tree t_{2r}. △

2.3 Language, finite, tree and pushdown automata

We define notions from the theory of string and tree languages similarly as they are defined in [3, 37, 13, 17, 33].

2.3.1 Language

A *language* over an alphabet $\mathcal{A}$ is a set of strings over $\mathcal{A}$. The set of all strings over $\mathcal{A}$ is denoted by $\mathcal{A}^*$. Set $\mathcal{A}^+$ is defined as $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. The m-fold concatenation of x for string $x \in \mathcal{A}^*$ with $x^0 = \varepsilon$ is denoted x^m, $m \geq 0$. Set x^* is defined as $\{x^m : m \geq 0\}$, x^+ as $\{x^m : m \geq 1\}$ and $x^* = x^+ \cup \{\varepsilon\}$.

2.3.2 Finite automata

A *nondeterministic finite automaton* (NFA) is a five-tuple $FA = (Q, \mathcal{A}, \delta, q_0, F)$, where Q is a finite set of *states*, $\mathcal{A}$ is an *input alphabet*, δ is a mapping from $Q \times \mathcal{A}$ into a set of finite subsets of Q, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is the set of final (accepting) states. A finite automaton FA is *deterministic* (DFA) if $\delta(q, a)$ has no more than one member for any $q \in Q$ and $a \in \mathcal{A}$, hence the δ mapping of the deterministic finite automaton is usually simplified to a mapping from $Q \times \mathcal{A}$ into q, where $q \in Q$. We note that the mapping δ is often illustrated by its transition diagram.

Every NFA can be transformed to an equivalent DFA [3]. The transformation constructs the states of the DFA as subsets of states of the NFA and selects only such accessible states (i.e. subsets). These subsets are called *d–subsets*. Although d–subsets are standard sets, they are often written in square brackets ([ ]) instead of in braces ({ }).

2.3.3 Tree automata

The set $T(\mathcal{A})$ of *ground terms* over the ranked alphabet $\mathcal{A}$ is the smallest set inductively defined in the following way:

1. $\mathcal{A}_0 \subseteq T(\mathcal{A})$,
2. if $p \geq 1$, $f \in \mathcal{A}_p$ and $t_1, \ldots, t_p \in T(\mathcal{A})$, then $f(t_1, \ldots, t_p) \in T(\mathcal{A})$.

Ground terms can be regarded as finite labelled ordered ranked *trees* in prefix notation, where each symbol of $f \in \mathcal{A}$ represents a node with label f, and the arguments are its children. Therefore, the notions of tree and ground term will be used interchangeably [43].

A *nondeterministic finite frontier to root tree automaton* (NFRTA) over a ranked alphabet $\mathcal{A}$ is a 4−tuple $M = (Q, \mathcal{A}, F, \Delta)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and Δ is a mapping from $f(q_1, q_2, \ldots, q_n)$ into a set of finite subsets of Q, where $f \in \mathcal{A}_n$, $n \geq 0$, and $q_1, q_2, \ldots, q_n \in Q$.

The finite frontier to root tree automaton is deterministic (DFRTA) if $\Delta(f, q_1, q_2, \ldots, q_n)$ has no more than one member for any $f \in \mathcal{A}_n$, $n \geq 0$, and $q_1, q_2, \ldots, q_n \in Q$, hence the Δ mapping of a deterministic finite frontier to root tree automaton is simplified to a mapping from $f(q_1, q_2, \ldots, q_n)$ into q.

Example 2.3.1. An example of a DFRTA over an alphabet containing constants $b0$ and $c0$, and binary symbol $a2$ is $M_{ta1} = (Q, \mathcal{A}, F, \Delta)$, where $Q = \{1, 2, 3\}$, $\mathcal{A} = \{a2, b0, c0\}$, $F = \{3\}$, and Δ contains these transition rules:

$$\begin{aligned} b0 &\rightarrow 1 \\ c0 &\rightarrow 2 \\ a2(1,1) &\rightarrow 3 \\ a2(1,2) &\rightarrow 3 \end{aligned}$$

The transition function of DFRTA can be depicted using a diagram. The automaton M_{ta1} is depicted in Figure 2.5. The transition function diagram of a frontier to root tree automaton is similar to the transition function diagram of a finite automaton. However, transitions of frontier to root tree automaton can have no source state, single source state, or even multiple source states. The order of source states is given by numbering the respective edge. △

DFRTAs over a ranked alphabet $\mathcal{A}$ runs on ground terms, respectively trees, over $\mathcal{A}$. The computation of DFRTA starts at leaves and moves towards the root. The automaton

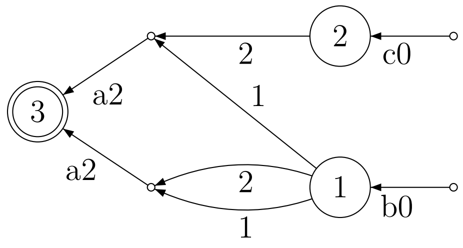


Figure 2.5: Visualisation of the DFRTA M_{ta1} from Example 2.3.1.

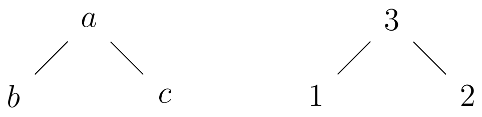


Figure 2.6: Tree t (left) and the run of DFRTA M_{ta1} from Example 2.3.1 on tree t (right).

maps each subterm, each subtree respectively, to a state, inductively. A *run* of an automaton is defined as follows:

1. The leaves are mapped to states q given by the transition rules of the form $a \rightarrow q \in \Delta$, where $a \in \mathcal{A}_0$.
2. Given a node labelled with $f \in \mathcal{A}_n$, $n \geq 1$, and its children mapped to states $q_1, \dots, q_n$ then this node is mapped to q, where $f(q_1, q_2, \dots, q_n) \rightarrow q \in \Delta$.

A ground term, or tree respectively, is *accepted* by a finite frontier to root tree automaton if there exists a run on the ground term, or tree, such that its root is mapped to a final state.

The tree language $L(\mathcal{A})$ *recognised* by a DFRTA $\mathcal{M}$ is the set of all ground terms, or trees respectively, accepted by the DFRTA $\mathcal{M}$. A tree language is *recognizable* if it is recognised by some NFRTA. A tree language is recognisable if and only if it is a regular tree language (see [13, 17, 33] for the definition of regular tree languages).

Two finite frontier to root tree automata, deterministic or nondeterministic, are equivalent if and only if they recognise the same language.

Every NFRTA can be transformed to an equivalent DFRTA [17].

Example 2.3.2. A tree language $L(M_{ta1}) = \{a2(b0, b0),\ a2(b0, c0)\}$ is recognised by the DFRTA M_{ta1}. The ground term $t = a2(b0, c0)$ and the run of DFRTA M_{ta1} on ground term t are illustrated in Figure 2.6. △

Note that the tree automata can also traverse the tree staring at root in the direction to leaves. Such a tree automaton is called *nondeterministic root to frontier finite tree automata* (NRFTA) and *deterministic top-down finite tree automata* (DRFTA), respectively.

In contrast to NFRTA, some NRFTAs are not possible to transform to an equivalent deterministic DRFTA. The NRFTA is strictly more powerful computation model than

DRFTA. On the other hand, the NRFTA, NFRTA, are DFRTA are all equally powerful computation models.

2.3.4 Pushdown automata

A *nondeterministic pushdown automaton* (nondeterministic 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 mapping 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.

Triple $(q, w, x) \in Q \times \mathcal{A}^* \times G^*$ denotes the configuration of a pushdown automaton. We will write the top of the pushdown store x 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 $(p, \gamma) \in \delta(q, a, \alpha)$. 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 is *input–driven* if each of its pushdown operations is determined only by the input symbol.

A language L accepted by a pushdown automaton M is defined in two distinct ways:

1. *Accepting by final state:* $L(M) = \{x : (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \gamma) \wedge x \in \mathcal{A}^* \wedge \gamma \in G^* \wedge q \in F\}$.
2. *Accepting by empty pushdown store:* $L_\varepsilon(M) = \{x : (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon) \wedge x \in \mathcal{A}^* \wedge q \in Q\}$.

If the pushdown automaton accepts the language by empty pushdown store, then the set F of final states is empty.

Unreachable states of automaton $M = (Q,\ \mathcal{A},\ G,\ \delta,\ q_0,\ Z_0,\ F)$ are states $p \in Q$ which are not reachable from the initial state with no sequence of transitions from the initial state to that particular state p. Formally, there are no transitions that allow $(q_0, kw, Z_0) \vdash_M^+ (p, w, \gamma)$.

Pushdown automaton $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ is acyclic if it does not contain transitions $(q, x_1, \gamma_1) \vdash_M^+ (q, x_2, \gamma_2)$, where $xx_2 = x_1$, $x \neq \varepsilon$ and $q \in Q$.

2.4 (Compact) string suffix and factor automata

String suffix and factor automata are finite automata that were introduced in [6, 18] as a mechanism for eliminating redundancy in string suffix trees [19, 23, 51, 58]. Given a string $s \in \mathcal{A}^*$, the suffix and factor automaton constructed for the string s represents an index of the string s capable of locating occurrences of its suffixes and substrings. The time needed to locate these occurrences of suffixes and substrings, of the string s, respectively, is linear with respect to the length of the located suffix and substring, respectively and the actual number of occurrences. The time does not depend on the length of the indexed string s.

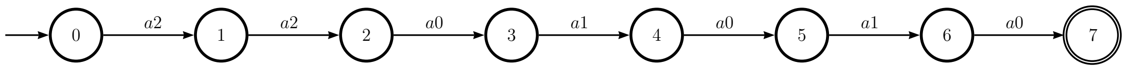


Figure 2.7: A transition diagram of the backbone of nondeterministic suffix automaton $FA_b(pref(t_{1r}))$ for prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ of tree t_{1r} from Example 2.2.1.

In [19, 23, 58], the suffix and factor automata are defined as minimal deterministic finite automata accepting the language of all suffixes and substrings of string s, respectively. In [53, 51], their basic nondeterministic versions are also presented. In some literature, the deterministic suffix automaton is also called the *directed acyclic word graph* (DAWG).

In order to construct a nondeterministic suffix and factor automaton for the string s a backbone of the nondeterministic suffix or factor automaton shall be constructed. The backbone of the suffix or factor automaton accepts a single string s. The automaton is a sequence of states connected by transitions each reading a symbol of the string s. The first state of the sequence is an initial state, and the last state is a final state.

Example 2.4.1. Given a string $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is the prefix notation of tree t_{1r} from Example 2.2.1, the corresponding backbone of the suffix automaton is $FA_b(pref(t_{1r})) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \delta_n, 0,\ \{7\})$, where its transition diagram is illustrated in Figure 2.7. △

The extension from the backbone of the suffix automaton for the string s is in addition of epsilon transitions from an initial state to all other states. The extension from the suffix automaton to factor is in change of all non-final states of the suffix automaton to final states.

The epsilon transitions need to be removed from the automaton by epsilon transitions removal algorithm, or direct construction of the automaton resulting from the epsilon transitions removal can be designed. The epsilon transitions removal is in this concrete case straightforward as the epsilon transitions do not form a sequence. For the formal construction of the nondeterministic suffix and factor automaton, see [53, 51].

Example 2.4.2. Given a string $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is the prefix notation of tree t_{1r} from Example 2.2.1, the corresponding nondeterministic suffix automaton (without epsilon transitions) is $FA_{nsuf}(pref(t_{1r})) = (\{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7\},\ \mathcal{A},\ \delta_n,\ 0,\ \{7\})$, where its transition diagram is illustrated in Figure 2.8. △

It is possible to compactise transitions as described in [21] to create a space-efficient version of the suffix automaton.

A *compact suffix automaton* [21] for a text is a finite automaton that accepts all suffixes of the text. The outgoing transitions of each state of the compact suffix automaton are labelled by some substring of the indexed string encoded by two indexes to the indexed string

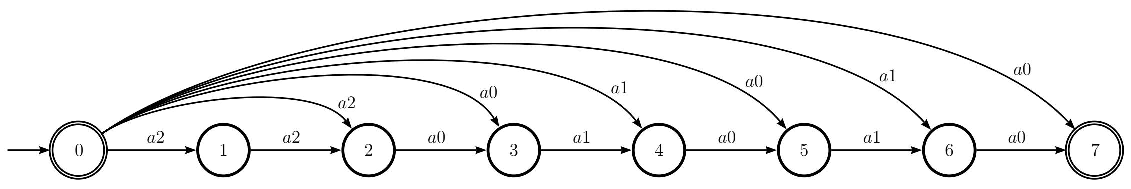


Figure 2.8: A transition diagram of nondeterministic suffix automaton $FA_{nsuf}\ (pref(t_{1r}))$ for prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ of tree t_{1r} from Example 2.2.1.

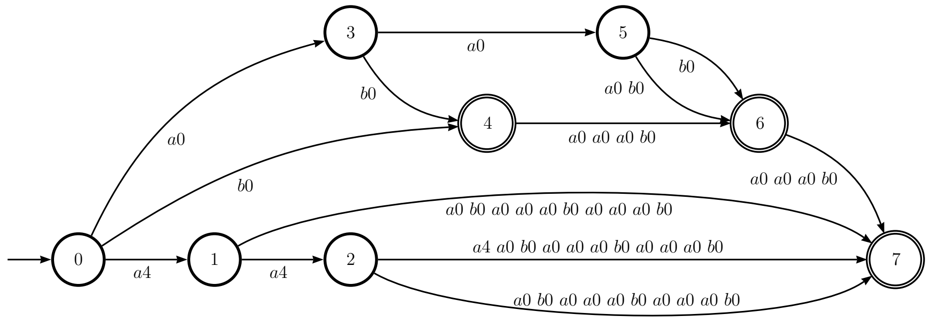

0
1
2
3
4
5
6
7
a4
a4
a0
b0
a0
b0
b0
a0 b0
a0 a0 a0 b0
a0 a0 a0 b0
a0 b0 a0 a0 a0 b0 a0 a0 a0 b0
a4 a0 b0 a0 a0 a0 b0 a0 a0 a0 b0
a0 b0 a0 a0 a0 b0 a0 a0 a0 b0

Figure 2.9: A transition diagram of compact suffix automaton $FA_{csuf}(pref(t_{2r}))$ for tree t_{2r} from Example 2.2.4. The long edge labels can be represented by pairs of beginning and ending indices into $pref(t_{2r})$, see [21].

$(start, end)$. All substring labels of outgoing transitions of individual states of the compact suffix automaton differ by their first symbol. Hence the automaton is deterministic.

An *explicit state* of the compact suffix automaton is a state that is represented in the automaton. An *implicit state* of the compact suffix automaton is a state not represented in the automaton that is between every two symbols on each transition.

By means of the compact suffix automaton, the list of positions of all occurrences of an input string pattern y of size m can be computed in time $\mathcal{O}(m + |occ(y)|)$, where $occ(y)$ is the set of all occurrences of y in the text.

A compact suffix automaton can also be constructed directly by a linear time algorithm presented in [25] and by a linear time online algorithm presented in [39].

Example 2.4.3. Given a string $pref(t_{2r}) = a4\ a4\ a4\ a0\ b0\ a0\ a0\ a0\ b0\ a0\ a0\ a0\ b0$, which is the prefix notation of tree t_{2r} from Example 2.2.4, the corresponding compact suffix automaton $FA_{csuf}(pref(t_{2r}))$ is illustrated in Figure 2.9. △

Algorithm 1 (FindOccurrences) can be used to query the compact suffix automaton. The accepting run of the automaton with a string pattern can end in an explicit state or an implicit state (between two states). The explicit state itself or the target state of the transition where the run ended is *reached* after processing the string pattern by the automaton.

For the future use of Algorithm 1 (FindOccurrences) is designed to return pairs of indexes representing the begin and end of each occurrence.

```
Name: FindOccurrences
Input: Compact suffix automaton $M$ constructed for a subject string $S$ of length
       $n$
Input: Pattern string $p$ of length $m$
Result: Occurrences $occ^S(P)$
1 begin
2     Let $q$ be the state of the automaton $M$ reached after processing $p$;
3     Find all paths from state $q$ that lead to a final state of $M$;
4     For each path of length $length$ from state $q$ to a final state, add occurrence
        $(n - length - m, n - length)$ of the string pattern $p$ to $occ^S(P)$;
5     return $occ^S(P)$;
6 end
```

Algorithm 1: Finding Occurrence of Patterns [21].

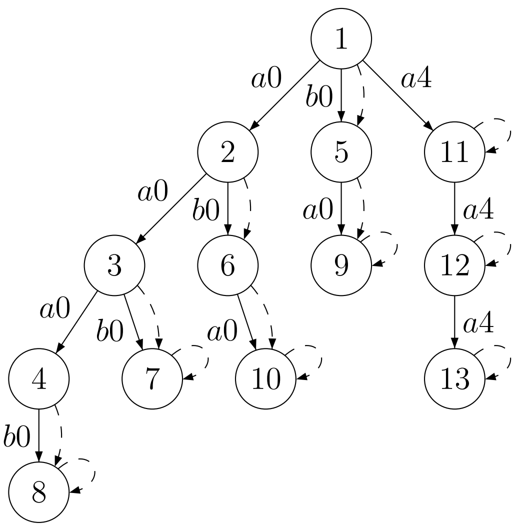
2.5 Position heap

A *position heap* is a tree-like structure introduced in [27]. Position heap itself is based on another structure – sequence tree [14]. Given a subject string $s \in \mathcal{A}^*$ of size n, the position heap constructed for the string s has size $O(n)$. The position heap queried with pattern string $p \in \mathcal{A}^*$ of size m reports all occurrences of the pattern p in the string s in $O(m + k)$ time, where k is the number of occurrences of pattern p. To achieve this query time complexities the position heap needs to be augmented (see [27] for details). The time bounds are the same as those of the compact suffix automaton and many other string indexing structures. Later in [47], another on-line construction algorithm of position heap was introduced. The on-line constructed position heap has the same size and allows queries to run in the same asymptotic time.

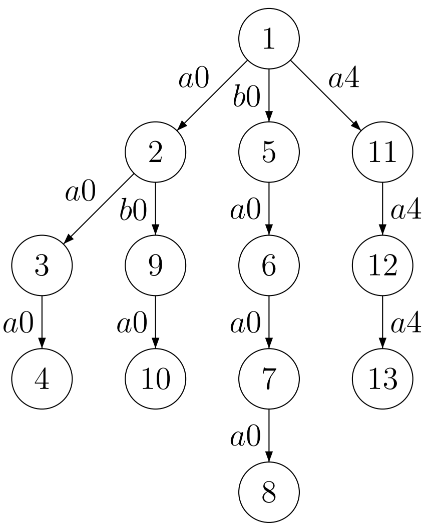
The naive construction algorithm presented in [27] iteratively adds all suffixes of the subject string shortest to longest to the created position heap. Adding each suffix introduces exactly one new node to the position heap. The new node represents an index to the subject uniquely identifying the respective suffix.

The linear time construction algorithm uses a separate *dual-heap* structure. The dual-heap structure stores the same string as the position heap however reversed. The dual-heap and position heap together help each other locate the parent node of the newly inserted node representing the new longer suffix.

Example 2.5.1. Given a string $pref(t_{2r}) = a4\ a4\ a4\ a0\ b0\ a0\ a0\ a0\ b0\ a0\ a0\ a0\ b0$, which is the prefix notation of tree t_{2r} from Example 2.2.4, the corresponding position heap $PH(pref(t_{2r}))$ and its dual-heap $DH(pref(t_{2r}))$ are illustrated in Figure 2.10. The



(a) A position heap $PH(pref(t_{2r}))$ for tree t_{2r} from Example 2.2.4.



(b) A dual heap $DH(pref(t_{2r}))$ used in construction of the position heap for tree t_{2r} from Example 2.2.4.

Figure 2.10: A position heap $PH(pref(t_{2r}))$ (left) and dual heap $DH(pref(t_{2r}))$ (right) constructed as indexes of tree t_{2r} from Example 2.2.4.

Table 2.2: Order of suffixes of subject tree t_{2r} in prefix notation as inserted to position heap $PH(pref(t_{2r}))$.

order	13	12	11	10	9	8	7	6	5	4	3	2	1
$pref(t_{2r})$	a4	a4	a4	a0	b0	a0	a0	a0	b0	a0	a0	a0	b0

suffixes of $pref(t_{2r})$ are inserted to the position heap $PH(pref(t_{2r}))$ in reversed order as illustrated in Table 2.2. △

Linear time queries require the position heap additionally augmented with *maximal-reach pointers* and discovery and finishing times of all nodes based on depth-first traversal of the position heap. Using the dual-heap the maximal reach pointers can also be constructed in $O(n)$ time.

The query algorithm uses the position heap to find pattern string p either by reaching node representing the pattern or by reaching a node representing the longest prefix of the pattern.

In the first case, the reached node and all its descendant nodes are occurrences of the pattern. Ancestor nodes need to be tested in $O(1)$ time whether they are occurrences as well using maximum reach pointers.

In the second case, the occurrences of the prefix of the pattern are computed by testing the reached node and its ancestor nodes similarly. However, the position heap is used again to find occurrences of the longest prefix of the remaining suffix of the pattern from the previous query. Resulting occurrences from the second query are used to filter out

Table 2.3: Subject tree t_{2r} from Example 2.2.4 in its prefix notation $pref(t_{2r})$, the suffix array and the longest prefix array constructed for subject tree t_{2r} in the prefix notation.

suffix array index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$pref(t_{2r})$	a4	a4	a4	a0	b0	a0	a0	a0	b0	a0	a0	a0	b0	\$
suffix array $SA(pref(t_{2r})$	14	10	6	11	7	12	8	4	3	2	1	13	9	5
longest common prefix array $LCP(pref(t_{2r}))$	-	0	4	2	3	1	2	6	0	1	2	0	1	5

occurrences from the first query. This querying and filtering is both repeated until some node of the position heap represent the complete remaining suffix of the pattern.

2.6 Suffix array

A *suffix array* is a linear structure introduced in [50]. Suffix array stores indexes to the indexed string lexicographically sorted to allow fast querying. Given a subject string $s \in \mathcal{A}^*$ of size n, the suffix array constructed for the string s has size $O(n)$. The suffix array was originally proposed to be constructed in $O(n \log(n))$ time, however, the expected construction time is shown to be $O(n)$ [46, 44, 54]. The suffix array queried with pattern string $p \in \mathcal{A}^*$ of size m reports all occurrences of the the pattern p in the string s in $O(m \log(n))$ time as proposed originally. The query time can be improved to $O(m + \log(n))$ utilising an additional *longest common prefix* structures.

An extra terminating symbol \$ is placed at the end of strings indexed by suffix arrays. The symbol is lexicographically smaller than any other and it is not used elsewhere in the string.

Example 2.6.1. Given a string $pref(t_{2r}) = a4\ a4\ a4\ a0\ b0\ a0\ a0\ a0\ b0\ a0\ a0\ a0\ b0$, which is the prefix notation of tree t_{2r} from Example 2.2.4, the corresponding suffix array $SA(pref(t_{2r}))$ and its longest common prefix array $LCP(pref(t_{2r}))$ are illustrated in Table 2.3. $\triangle$

Many other construction algorithms and representations of suffix arrays were presented. Construction algorithms were improved to be at worst truly linear with respect to the size of the indexed string. The representation of suffix arrays can be improved up to $5n$ bytes. For more details see [56].

Combining the suffix array and the longest prefix array with a *child table*, the querying time can be further improved to the optimal $O(m)$ as shown in [1]. The additional child table allows navigating in the suffix array as if it were a suffix tree. Such a possibility effectively allows any computation on suffix tree to be directly translated to computation on suffix array instead.

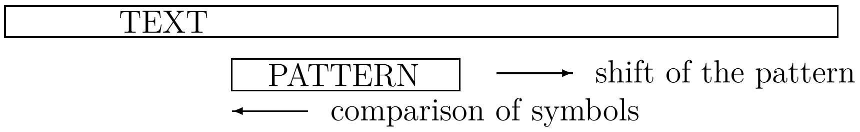


Figure 2.11: Graphical outline of Algorithm 2 (BasicBackwardPatternMatchingAlgorithm).

2.7 Backward string pattern matching algorithm

The basic backward (string) pattern matching algorithm featuring only the different direction of symbol comparison and pattern shift is presented as Algorithm 2 (BasicBackwardPatternMatchingAlgorithm). No preprocessing of the pattern or subject is needed. The matching time with the basic backward pattern matching algorithm is $O(m * n)$, where m is the size of the pattern and n is the size of the subject. The basic backward pattern matching algorithm is a common base for many modifications, which make it more efficient in practice.

```
Name: BasicBackwardPatternMatchingAlgorithm
Input: A string text of size $n$ and a string pattern of size $m$
Result: Locations of the pattern in the text
1  begin
2      $i := 0$;
3      while $i < n - m$ do
4          $j := m$;
5          while $j > 0$ and $pattern[j] = text[i + j]$ do
6              $j := j - 1$;
7          end
8          if $j = 0$ then yield $i + 1$;
9          $i := i + 1$; {Length of the shift.}
10     end
11 end
```

Algorithm 2: Basic backward string pattern matching algorithm.

Instead of a shift by 1 (as per line 9 of Algorithm 2 (BasicBackwardPatternMatchingAlgorithm)), longer shifts can be made.

The first algorithm from the category of backward (string) pattern matching algorithms was introduced in [7] (nowadays known as the Boyer-Moore algorithm) to further decrease the number of comparisons of symbols of the pattern and subject. The number of comparisons is expected to be sublinear in contrast to the state-of-the-art algorithms of that time such as [45] with at best feature linear number of comparisons. Given a string $s \in \mathcal{A}^*$ referred to as subject or text and a string $p \in \mathcal{A}^*$ referred to as a pattern, the occurrences of the pattern p in the text s are located by comparing the symbols of the pattern and the text in the opposite direction to the shifting of the pattern.

Another algorithm from the category of backward pattern matching algorithms is the Boyer-Moore-Horspool algorithm [38]. The shift table of the Boyer-Moore-Horspool algorithm is represented by a *bad character shift table.* The Boyer-Moore-Horspool algorithm is the evolution of the Boyer-Moore algorithm. It computes the length of the shift based on one symbol aligned to the end of the pattern. This shift has turned out to perform very well in practice despite being a simplification of the heuristics used by the original Boyer-Moore algorithm.

Definition 2.7.1. Let $pattern[1..m]$ be a pattern of size m over an alphabet $\mathcal{A}$. The bad character shift table $BCS(pattern[1..m]$ is defined for each $a \in \mathcal{A}$ as follows:

$$BCS(pattern[1..m])[a] = \min(\{m\} \cup \{j : pattern[m-j] = a \text{ and } m > j > 0\})$$

The definition is an adaptation from [20]. The definition is adapted to reflect indexing of strings from one to their length, respectively.

Example 2.7.2. Consider a string $pref(p_{1r}) = a2\ a0\ a1\ a0$ over an alphabet $\mathcal{A} = \{a3,$ $a2,\ a1,\ a0\}$, which is a prefix notation of a tree pattern p_{1r} depicted in Figure 2.3. The $BCS(pref(p_{1r}))$ abbreviated as BCS contains the following items: $\triangle$

$$\begin{array}{ll} BCS[a3] = \min(\{4\} \cup \emptyset) = 4 & BCS[a2] = \min(\{4\} \cup \{3\}) = 3 \\ BCS[a1] = \min(\{4\} \cup \{1\}) = 1 & BCS[a0] = \min(\{4\} \cup \{2\}) = 2 \end{array}$$

Chapter 3

Previous Results and Related Work

In this chapter, algorithms and theorems regarding tree pattern pushdown automata for trees in the prefix notation are given, and the tree pattern PDAs and their construction are demonstrated on an example. A tree pattern can be either a subtree or a tree template, which contains at least one wildcard symbol S representing a subtree. Tree pattern PDAs are an extension of subtree PDAs, introduced in [41]. A subtree PDA is analogous to the string suffix automaton and it accepts a linear notation of all subtrees of a given tree. The pushdown operations are used to process the tree structure. New states and transitions, which are used for processing the wildcard symbols S in tree templates, are additionally present in the tree pattern PDA. The pushdown operations are the same. The tree pattern pushdown automata are introduced in [51, 52].

3.1 Indexing trees for subtree matching

A subtree pushdown automata are designed as a structure for indexing trees for fast searching of subtree locations in the indexed tree. The structure was introduced in [41].

Definition 3.1.1. Let t and $pref(t)$ be a tree and its prefix notation, respectively. A *subtree pushdown automaton* for $pref(t)$ accepts all subtrees in the prefix notation which match tree t.

Given a subject tree t first an automaton accepting single string $pref(t)$ shall be constructed. This automaton represents a *backbone* of the subtree pushdown automaton. The automaton can be constructed by Algorithm 3 (ConstructSubtreePDABackbone). The correctness Algorithm 3 (ConstructSubtreePDABackbone) is proved by Theorem 3.1.3.

The backbone of subtree PDA is by itself a deterministic pushdown automaton.

Example 3.1.2. Consider tree t_{1r} in prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 2.2.1, which is illustrated in Figure 2.1a. The deterministic backbone of the subtree PDA, constructed by Algorithm 3 (ConstructSubtreePDABackbone) is determin-

```
Name: ConstructSubtreePDABackbone
Input: A tree $t$ over a ranked alphabet $\mathcal{A}$ in prefix notation $pref(t) = a_1a_2\dots a_n$,
       $n \geq 1$
Result: Backbone of subtree PDA $M_p(t) = (\{0,1,2,\dots,n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$
1 begin
2     Create $M_p(t)$ as an automaton with initial state 0 and states $0 \leq i \leq n$;
3     foreach state $i$, where $1 \leq i \leq n$ do
4         Add a new transition $\delta(i-1, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$ to
           automaton $M_p(t)$;
5     end
6     return $M_p(t)$;
7 end
```

Algorithm 3: Construction of a backbone of the subtree PDA $M_p(t)$ for a tree t in prefix notation $pref(t)$.

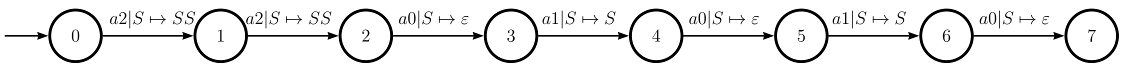

$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
0
1
2
3
4
5
6
7

Figure 3.1: A transition diagram of deterministic backbone of subtree PDA $M_p\ (t_{1r})$ for tree t_{1r} in prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 3.1.2.

istic PDA $M_p(t_{1r}) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \{S\}, \delta_p, 0, S, \emptyset)$, where mapping δ_p is a set of the following transitions:

$$\begin{array}{ll}
\delta_p(0, a2, S) = (1, SS) & \delta_p(1, a2, S) = (2, SS) \\
\delta_p(2, a0, S) = (3, \varepsilon) & \delta_p(3, a1, S) = (4, S) \\
\delta_p(4, a0, S) = (5, \varepsilon) & \delta_p(5, a1, S) = (6, S) \\
\delta_p(6, a0, S) = (7, \varepsilon) &
\end{array}$$

Note that the pushdown automaton accepts a string by an empty pushdown store.

The transition diagram of deterministic backbone of subtree PDA $M_p(t_{1r})$ is illustrated in Figure 3.1. For each transition $\delta(p, a, \alpha) = (q, \beta)$ from δ the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \mapsto \beta$. △

Theorem 3.1.3. Given a tree t and its prefix notation $pref(t)$, the PDA $M_p(t)$ constructed by Algorithm 3 (ConstructSubtreePDABackbone) accepts only a single string $pref(t)$.

Proof. The automaton omitting the pushdown store operations is a simple finite automaton, which starting in its initial state can only read string $pref(t)$. The original pushdown automaton accepts by emptying the pushdown store and the pushdown store operations created by the Algorithm 3 (ConstructSubtreePDABackbone) operates the height of the pushdown store so that it effectively simulates the computation of arity checksum. Since the sequence of transitions represented by the automaton was constructed for a complete tree, the only state where the pushdown store is empty is the last one. Therefore the automaton can only accept when reaching the last state by reading the whole string $pref(t)$. □

The pushdown store operations seem unnecessary, however that are necessary for more complex variants of pushdown automata accepting subtrees, tree pattern, and nonlinear tree patterns.

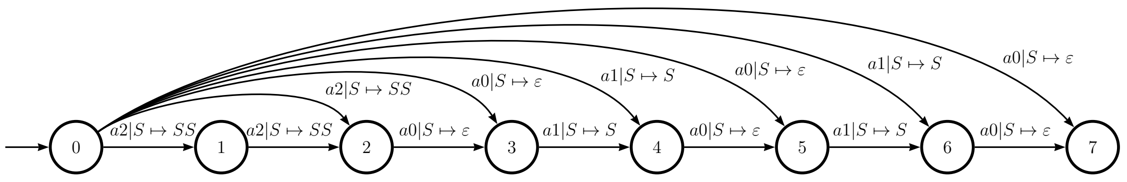
Similarly as the backbone of the nondeterministic suffix automaton for string s is extended to nondeterministic suffix automaton, the backbone of the subtree PDA can be extended with transitions from initial state to every other state. The label on these new transitions is shared with already existent transitions on the backbone where the target state of the new transition and a transition from the backbone is the same. Note that the label includes the pushdown store manipulation. The resulting pushdown automaton accepts a string by an empty pushdown store so there are no final states added. The following algorithm formally describes the construction of the subtree pushdown automaton.

```
Name: ConstructSubtreePDA
Input: A tree $t$ over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1 a_2 \dots a_n$,
       $n \geq 1$
Result: Nondeterministic subtree PDA $M_{nps}(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A} \cup \{S\},$
$\{S\}, \delta, 0, S, \emptyset)$
1 begin
2     Create $M_{nps}(t)$ as $M_p(t)$ by Algorithm 3 (ConstructSubtreePDABackbone);
3     foreach state $i$, where $2 \leq i \leq n$ do
4         create a new transition $\delta(0, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$ in
            automaton $M_{nps}(t)$;
5     end
6     return $M_{nps}(t)$;
7 end
```

Algorithm 4: Construction of a nondeterministic subtree PDA for a tree t in prefix notation $pref(t)$.

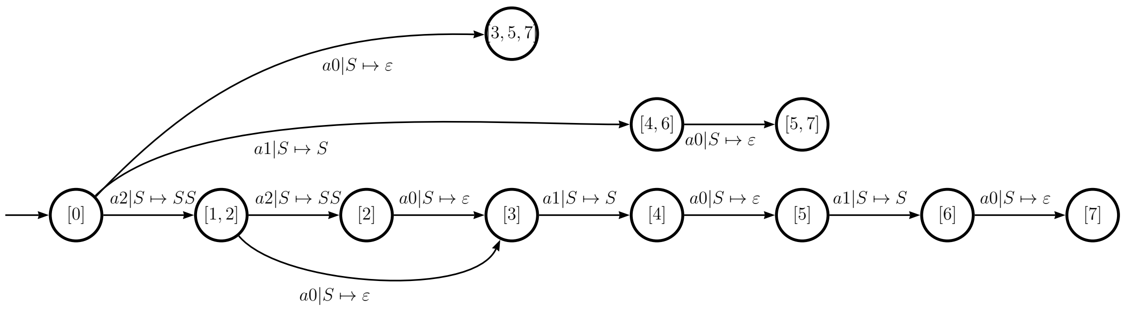
Example 3.1.4. Consider tree t_{1r} in prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 2.2.1, which is illustrated in Figure 2.1a. The nondeterministic subtree PDA accepting all subtrees of tree t_{1r}, which has been constructed by Algorithm 4 (ConstructSubtreePDA), is nondeterministic PDA $M_{nps}(t_{1r}) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \{S\}, \delta_2, 0, S, \emptyset)$, where mapping δ_2 is a set of the following transitions:

$$\begin{array}{ll}
\delta_2(0, a2, S) = (1, SS) & \\
\delta_2(1, a2, S) = (2, SS) & \delta_2(0, a2, S) = (2, SS) \\
\delta_2(2, a0, S) = (3, \varepsilon) & \delta_2(0, a0, S) = (3, \varepsilon) \\
\delta_2(3, a1, S) = (4, S) & \delta_2(0, a1, S) = (4, S) \\
\delta_2(4, a0, S) = (5, \varepsilon) & \delta_2(0, a0, S) = (5, \varepsilon) \\
\delta_2(5, a1, S) = (6, S) & \delta_2(0, a1, S) = (6, S) \\
\delta_2(6, a0, S) = (7, \varepsilon) & \delta_2(0, a0, S) = (7, \varepsilon)
\end{array}$$



0
1
2
3
4
5
6
7
$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$

Figure 3.2: A transition diagram of nondeterministic subtree pushdown automaton M_{nps} (t_{1r}) from Example 3.1.4 for tree t_{1r} in prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 2.2.1.



[0]
[1, 2]
[2]
[3]
[4]
[5]
[6]
[7]
[3, 5, 7]
[4, 6]
[5, 7]
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$

Figure 3.3: A transition diagram of deterministic subtree pushdown automaton M_{dps} (t_{1r}) from Example 3.1.4 for tree t_{1r} in prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 2.2.1.

The transition diagram of nondeterministic subtree PDA $M_{nps}(t_{1r})$ is illustrated in Figure 3.2. Again, for each transition $\delta(p, a, \alpha) = (q, \beta)$ from δ the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \mapsto \beta$.

The deterministic subtree pushdown automaton constructed from nondeterministic subtree pushdown automaton $M_{nps}(pref(t_{1r}))$, is deterministic pushdown automaton M_{dps} $(pref(t_{1r})) = (\{[0], [1,2], [2], [3], [4], [5], [6], [7], [3,5,7], [4,6], [5,7]\}, \mathcal{A}, \{S\}, \delta_3, [0], S, \emptyset)$. Its transition diagram is illustrated in Figure 3.3.

Some states of deterministic subtree automaton always have an empty pushdown store when active in any run of the automaton. In this such states are $[3,5,7]$ and $[5,7]$. Therefore transitions leading from these states can be omitted. It means that the deterministic subtree pushdown automaton $M_{dps}(t_{1r})$ shown in Figure 3.3 has fewer transitions than the deterministic string suffix automaton constructed for $pref(t_{1r})$ [19, 51, 58]. △

Theorem 3.1.5. Given a tree t and its prefix notation $pref(t)$, the PDA $M_{nps}(t)$ constructed by Algorithm 4 (ConstructSubtreePDA) accepts all subtrees of the tree t in the prefix notation.

Proof. The automaton omitting the pushdown store operations and with all states final is a nondeterministic factor automaton, which starting in its initial state can read all factors of the string $pref(t)$. The original pushdown automaton accepts by emptying the pushdown store and the pushdown store operations created by the Algorithm 4 (ConstructSubtreePDA) operates the height of the pushdown store so that it effectively simulates the computation of arity checksum. Subtrees of a tree represent complete trees themselves. The pushdown store of the subtree pushdown automaton is therefore emptied after reading a complete tree. Hence the automaton can only accept after reading a complete tree and after reading a factor of the prefix notation of a tree. The combination of conditions exactly corresponds to the subtrees of the tree. □

3.2 Indexing trees for tree pattern matching

A tree pattern pushdown automata are designed as a structure for indexing trees for fast searching of a tree pattern locations in the indexed tree. The structure was introduced in [51, 52].

Definition 3.2.1. Let t and $pref(t)$ be a tree and its prefix notation, respectively. A *tree pattern pushdown automaton* for $pref(t)$ accepts all tree patterns in the prefix notation which match the tree t.

Given a subject tree in the prefix notation, first, so-called deterministic *treetop PDA* is constructed for this tree. The treetop PDA accepts all tree patterns that match the subject tree and contain the root of the subject tree. The deterministic treetop PDA is defined by the following definition. States and transitions of the treetop pushdown automaton are computed by Algorithm 5 (ConstructTreetopPDA). Finally, the correctness Algorithm 5 (ConstructTreetopPDA) is proved by Theorem 3.2.4.

Definition 3.2.2. Let t, r and $pref(t)$ be a tree, its root and its prefix notation, respectively. A *treetop pushdown automaton* $M_{pt}(t) = (0, 1, 2, \ldots, n, A \cup S, S, \delta, 0, S, \emptyset)$ for $pref(t)$ accepts all tree patterns in the prefix notation which contain the root r and match the tree t.

The construction of the treetop PDA is described by the following algorithm. The treetop PDA is deterministic.

Note that the abbreviation *srms* stands for Subtree Right Most States.

The treetop PDA is similar to the prefix string finite automaton or the backbone of the suffix or factor automaton. Moreover, there exist additional transitions reading symbol S, which represent subtrees. These transitions skip over parts of the prefix notation of the subject tree which are its subtrees. The automaton uses the pushdown store for computing a checksum so that the input would be a valid prefix notation of a tree.

The construction of treetop PDA by Algorithm 5 (ConstructTreetopPDA) is illustrated in the following example.

Example 3.2.3. Consider tree t_{1r} in prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 2.2.1, which is illustrated in Figure 2.1a. The deterministic treetop PDA, constructed by Algorithm 5 (ConstructTreetopPDA), is deterministic PDA $M_{pt}(t_{1r}) =$

```
Name: ConstructTreetopPDA
Input: A tree $t$ over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1a_2 \ldots a_n$,
       $n \geq 1$
Result: Treetop PDA $M_{pt}(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A} \cup \{S\}, \{S\}, \delta, 0, S, \emptyset)$
1  begin
2      Create $M_{pt}(t)$ as $M_p(t)$ by Algorithm 3 (ConstructSubtreePDABackbone);
3      Create a set $srms = \{ i \ : 1 \leq i \leq n,\ \delta(i-1, a, S) = (i, \varepsilon),\ a \in \mathcal{A}_0\}$;
4      for state $i = n-1$ downto 1, where $\delta(i, a, S) = (i+1, S^p)$, $a \in \mathcal{A}_p$ do
5          if $p = 0$ then
6              Create a new transition $\delta(i, S, S) = (i+1, \varepsilon)$;
7          else
8              Create a new transition $\delta(i, S, S) = (l, \varepsilon)$, where $l$ is the $p$-th smallest
                integer such that $l \in srms$ and $l > i$;
9              Remove all $j$, where $j \in srms$, and $i < j < l$, from $srms$;
10         end
11     end
12     return $M_{pt}(t)$;
13 end
```
Algorithm 5: Construction of a treetop PDA for a tree t in prefix notation $pref(t)$.

$(\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A} \cup \{S\}, \{S\}, \delta_1, 0, S, \emptyset)$, where mapping δ_1 is a set of the following transitions:

$$\begin{array}{ll}
\delta_1(0, a2, S) = (1, SS) & \\
\delta_1(1, a2, S) = (2, SS) & \delta_1(1, S, S) = (5, \varepsilon) \\
\delta_1(2, a0, S) = (3, \varepsilon) & \delta_1(2, S, S) = (3, \varepsilon) \\
\delta_1(3, a1, S) = (4, S) & \delta_1(3, S, S) = (5, \varepsilon) \\
\delta_1(4, a0, S) = (5, \varepsilon) & \delta_1(4, S, S) = (5, \varepsilon) \\
\delta_1(5, a1, S) = (6, S) & \delta_1(5, S, S) = (6, \varepsilon) \\
\delta_1(6, a0, S) = (7, \varepsilon) & \delta_1(6, S, S) = (7, \varepsilon)
\end{array}$$

The transition diagram of deterministic treetop PDA $M_{pt}(t_{1r})$ is illustrated in Figure 3.4. Again, for each transition $\delta(p, a, \alpha) = (q, \beta)$ from δ the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \mapsto \beta$.

Deterministic treetop PDA $M_{pt}(t_{1r})$ has been constructed by Algorithm 5 (ConstructTreetopPDA) manipulating the srms as follows. The *srms* is initially set to $\{3, 5, 7\}$. Then, new transitions, which read symbol S, are created using the set *srms* in the following order:

$$\delta_4(6, S, S) = (7, \varepsilon), \delta_4(5, S, S) = (7, \varepsilon), \delta_4(4, S, S) = (5, \varepsilon),$$
$$\delta_4(3, S, S) = (5, \varepsilon), \delta_4(2, S, S) = (3, \varepsilon), \text{and } \delta_4(1, S, S) = (5, \varepsilon)$$

The *srms* set is modified by removal of value 3 resulting in $srms = \{5, 7\}$ when the transition $\delta_4(1, S, S) = (5, \varepsilon)$ is created. This corresponds to a finalisation of computation inside the subtree represented by transitions on the backbone of the automaton between states 1 and 5. △

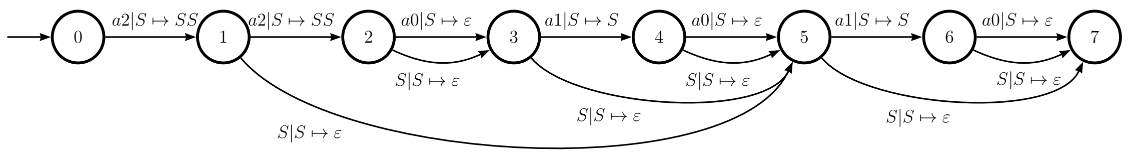

0
1
2
3
4
5
6
7
$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$

Figure 3.4: A transition diagram of deterministic treetop PDA M_{pt} (t_{1r}) from Example 3.2.3 for tree t_{1r} in prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 2.2.1.

Theorem 3.2.4. Given a tree t and its prefix notation $pref(t)$, the PDA $M_{pt}(t)$ constructed by Algorithm 5 (ConstructTreetopPDA) is a treetop PDA for $pref(t)$.

Proof. Let r be the root of t. The PDA $M_{pt}(t)$ is a simple extension of the PDA, which is constructed on line 1 of Algorithm 5 (ConstructTreetopPDA) and accepts the tree t in its prefix notation. New transitions, which read the wildcard symbol S that are added in the loop on lines 4 to 11 of Algorithm 5 (ConstructTreetopPDA). For these new transitions it holds that $\delta(q_1, S, S) = (q_2, \varepsilon)$ if and only if $(q_1, w, S) \vdash^+_{M_{pt}(t)} (q_2, \varepsilon, \varepsilon)$ and q_1 is not the initial state 0. This means that the new added transitions reading S correspond just to subtrees not containing the root r. Thus, the PDA $M_{pt}(t)$ accepts all tree patterns in the prefix notation which contain the root r and match the tree t. □

The nondeterministic tree pattern PDA for trees in the prefix notation is constructed as an extension of the deterministic treetop PDA. The extension is so that for each state of the treetop PDA with an incoming transition which reads a symbol $a \in \mathcal{A}$ we duplicate the transition and redirect it from the starting state to that state. This construction is described by the following algorithm.

```
Name: ConstructNTPPDA
Input: A tree $t$ over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1a_2 \dots a_n$,
       $n \geq 1$
Result: Nondeterministic tree pattern PDA $M_{npt}(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A} \cup \{S\},$
        $\{S\}, \delta, 0, S, \emptyset)$
1 begin
2     Create $M_{npt}(t)$ as $M_{pt}(t)$ by Algorithm 5 (ConstructTreetopPDA);
3     foreach state $i$, where $2 \leq i \leq n$ do
4         Create a new transition $\delta(0, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$ in
          automaton $M_{npt}(t)$;
5     end
6     return $M_{npt}(t)$;
7 end
```
Algorithm 6: Construction of a nondeterministic tree pattern PDA for a tree t in prefix notation $pref(t)$.

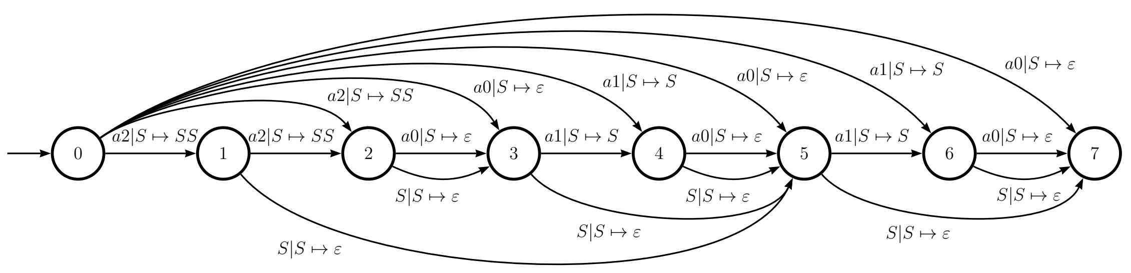

0
1
2
3
4
5
6
7
$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$

Figure 3.5: A transition diagram of nondeterministic tree pattern pushdown automaton M_{npt} (t_{1r}) from Example 3.2.5 for tree t_{1r} from Example 2.2.1 in prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$.

The tree pattern PDA is similar to the string factor finite automaton. Its construction is based on the treetop PDA and the extension is that the tree pattern rooted by the automaton can be matched on any node of the tree. For this reason, additional transitions are created.

Example 3.2.5. Consider tree t_{1r} in prefix notation $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 2.2.1, which is illustrated in Figure 2.1a. The nondeterministic tree pattern PDA accepting all tree patterns matching tree t_{1r}, which has been constructed by Algorithm 6 (ConstructNTPPDA), is nondeterministic PDA $M_{npt}(t_{1r}) = (\{0,1,2,3,4,5,6,7\}, \mathcal{A}, \{S\}, \delta_2, 0, S, \emptyset)$, where mapping δ_2 is a set of the following transitions:

$$\begin{array}{lll}
\delta_2(0,a2,S) = (1,SS) & & \\
\delta_2(1,a2,S) = (2,SS) & \delta_2(1,S,S) = (5,\varepsilon) & \delta_2(0,a2,S) = (2,SS) \\
\delta_2(2,a0,S) = (3,\varepsilon) & \delta_3(2,S,S) = (3,\varepsilon) & \delta_2(0,a0,S) = (3,\varepsilon) \\
\delta_2(3,a1,S) = (4,S) & \delta_2(3,S,S) = (5,\varepsilon) & \delta_2(0,a1,S) = (4,S) \\
\delta_2(4,a0,S) = (5,\varepsilon) & \delta_2(4,S,S) = (5,\varepsilon) & \delta_2(0,a0,S) = (5,\varepsilon) \\
\delta_2(5,a1,S) = (6,S) & \delta_2(5,S,S) = (6,\varepsilon) & \delta_2(0,a1,S) = (6,S) \\
\delta_2(6,a0,S) = (7,\varepsilon) & \delta_2(6,S,S) = (7,\varepsilon) & \delta_2(0,a0,S) = (7,\varepsilon)
\end{array}$$

The transition diagram of nondeterministic tree pattern PDA $M_{npt}(t_{1r})$ is illustrated in Figure 3.5. Again, for each transition $\delta(p,a,\alpha) = (q,\beta)$ from δ the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \mapsto \beta$.

The deterministic tree pattern PDA $M_{dpt}(t_{1r}) = (\{[0], [1,2], [2], [3], [4], [5], [6], [7], [3,5,7], [3,5], [4,6], [5,7]\}, \mathcal{A}, \{S\}, \delta_6, [0], S, \varnothing)$. Its transition diagram is illustrated in Figure 3.6.

Some states of the deterministic subtree automaton always have an empty pushdown store, therefore, some transitions can be omitted due to the pushdown operation.

The states where the pushdown store is always empty are states $[3,5,7]$ and $[5,7]$, therefore, all transitions leading from them can be omitted. △

In the following theorem we prove the correctness of the constructed tree pattern PDA.

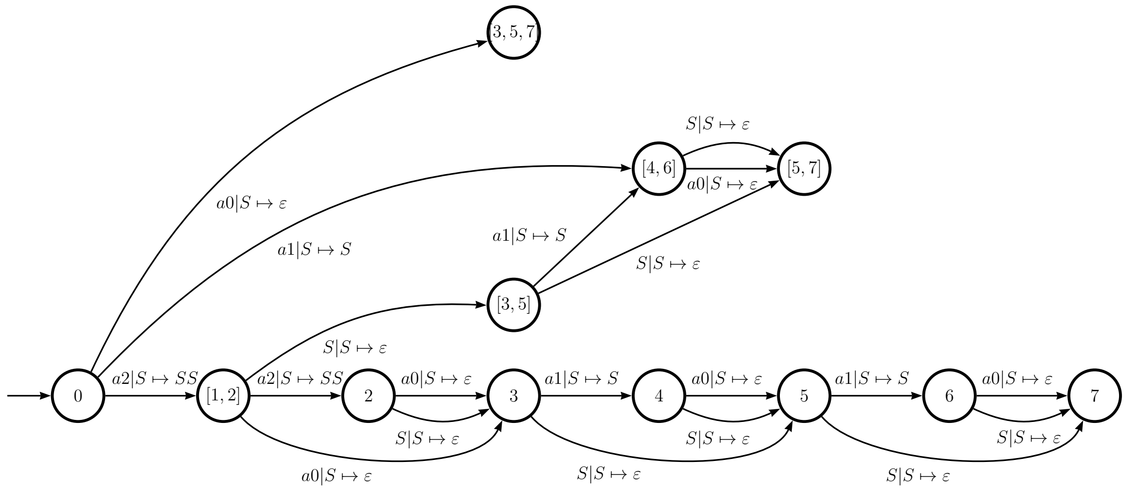

[3, 5, 7]
$S|S \mapsto \varepsilon$
[4, 6]
[5, 7]
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a1|S \mapsto S$
$S|S \mapsto \varepsilon$
[3, 5]
$S|S \mapsto \varepsilon$
0
$a2|S \mapsto SS$
[1, 2]
$a2|S \mapsto SS$
2
$a0|S \mapsto \varepsilon$
3
$a1|S \mapsto S$
4
$a0|S \mapsto \varepsilon$
5
$a1|S \mapsto S$
6
$a0|S \mapsto \varepsilon$
7
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$

Figure 3.6: A transition diagram of deterministic tree pattern pushdown automaton M_{dpt} (t_{1r}) from Example 3.2.5 for tree t_{1r} from Example 2.2.1 in prefix notation $pref(t_{1r}) =$ $a2\ a2\ a0\ a1\ a0\ a1\ a0$.

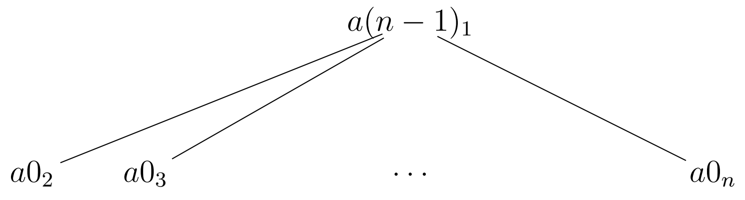
Theorem 3.2.6. Given a tree t and its prefix notation $pref(t)$, the PDA $M_{npt}(t)$ constructed by Algorithm 6 (ConstructNTPPDA) is a tree pattern PDA for $pref(t)$.

Proof. The PDA $M_{npt}(t)$ is a simple extension of the PDA $M_{pt}(t)$. The PDA $M_{pt}(t)$ is constructed by Algorithm 5 (ConstructTreetopPDA) and accepts all tree patterns in the prefix notation which contain the root r of the tree t and match the tree t by an empty pushdown store. The PDA $M_{npt}(t)$ contains newly added transitions of the form $\delta(0, a_i, S)$ $= (i, S^{Arity(a_i)})$. These transitions correspond just to the possibility that the first symbol of a tree pattern to be accepted can be any node of the tree t. Thus, the PDA $M_{npt}(t)$ accepts all tree patterns in the prefix notation which match the tree t. $\square$

Lemma 3.2.7. Given a tree t with n nodes, the number of distinct tree patterns which match the tree t can be at most $2^{n-1} + n - 1$.

Proof. First, subtrees of any subtree of the tree t can be replaced by the wildcard symbol S and the tree template resulting from such a replacement is a tree pattern which matches the tree. Given a tree with n nodes, the maximal number of subsets of subtrees that can be replaced by the wildcard symbol S occurs for the case of a tree t_{3r} whose structure is given by the prefix notation $pref(t_{3r}) = a(n-1)\ a0\ a0 \dots a0$, where $n \geq 2$. Such a tree is illustrated in Figure 3.7. In this tree, each of the $n-1$ nullary symbols $a0$ can be replaced by wildcard symbol S, and therefore we can create 2^{n-1} distinct tree templates which are tree patterns matching the tree t_{3r}.

Second, the subtrees of tree t are tree patterns which match the tree, which gives $n-1$ other distinct tree patterns (provided all the subtrees are unique).



$$pref(t_{3r}) = a(n-1)\ a0\ a0 \dots a0$$

Figure 3.7: A tree t_{3r} with $2^{n-1} + n$ distinct tree patterns matching the tree t_{3r} and its prefix notation.

Thus, the total number of distinct tree patterns matching the tree t can be at most $2^{n-1} + n - 1$. □

Lemma 3.2.8. Given a tree t in prefix notation $pref(t)$, the number of states of a nondeterministic tree pattern pushdown automaton $M_{npt}(t)$ is $m+1$, where m is the number of nodes of a subject tree.

Proof. There is one state for each symbol in $pref(t)$ plus the initial state. Thus, the number of states is $m+1$. □

Lemma 3.2.9. Given a tree t in prefix notation pref(t), the number of transitions of a nondeterministic tree pattern pushdown automaton $M_{npt}(t)$ is $3m-2$, where m is the number of nodes of a subject tree.

Proof. There is one transition for each symbol in $pref(t)$, which forms the backbone of the automaton. There are exactly $m-1$ transitions from the initial state to every other state. Finally, there is one transition for symbol S leading from every state except the initial state. Thus, the number of transitions is then $3m-2$. □

3.3 Tree pattern matching with tree automata

In this section, algorithms and theorems regarding tree pattern matching finite tree automata for trees are given. The tree pattern matching, finite tree automata and its construction are demonstrated on an example. NFRTAs accepting a tree pattern were introduced in [36] and used in [35, 34, 10, 29]. A taxonomy of tree pattern matching is given in [13].

A tree pattern matching NFRTA is similar to the string pattern matching NFA. It can implement various types of matching of trees or tree patterns. It can be constructed to accept all trees that match the tree pattern at the root. The tree automaton can also be constructed to locate all nodes rooting a subtree of a given tree that match given tree pattern. A tree pattern can be either a subtree or a tree template, which contains at least one special wildcard symbol S representing any subtree.

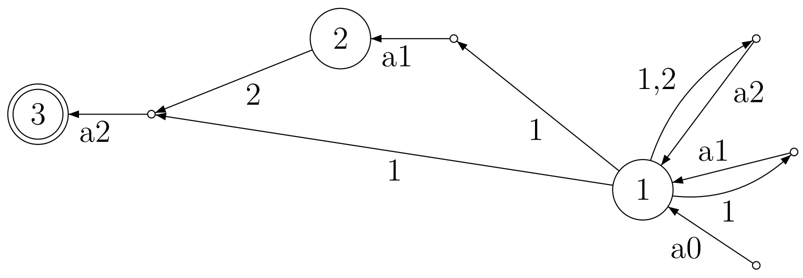


Figure 3.8: Visualisation of the tree pattern matching FTA $M_{ta2}(p_{2r})$ from Example 3.3.1.

Given a tree pattern p, the pattern matching on tree t can be reduced to assigning a set of subtrees of p to each node q of tree t. The assigned sets represent subtrees of p that match subtree of t rooted at node q. The number of such sets is finite and therefore can be precomputed.

The states of a deterministic finite frontier to root tree automaton constructed for tree pattern matching are equivalent to these sets and the transition function Δ of the automaton is given by all possible reductions of sets $q_1, q_2, \ldots, q_n$, assigned to nodes $s_1, s_2, \ldots, s_n$, to set q_f assigned to node f, where $f(s_1, s_2, \ldots, s_n)$ is a node of t and $s_1, s_2, \ldots, s_n$ are the children of f. The final states of the automaton are those equivalent to a set containing the complete pattern p.

The same result can be obtained by constructing a nondeterministic frontier to root tree automaton with subsequent determinisation.

The states of a nondeterministic finite frontier to root tree automaton are equivalent to subtrees of the pattern and Δ function of the automaton given by the structure of the pattern. The final state of the automaton is the one equivalent to the complete pattern p. The wildcard present in the pattern can be represented by a state of the automaton able to accept any tree over the alphabet of the subject tree.

Example 3.3.1. Given a tree pattern p_{2r} from Example 2.2.5 and nondeterministic frontier to root tree automaton $M_{ta}(p_{2r})$ constructed as the tree pattern matching automaton $M_{ta2}(p_{2r}) = (Q, \mathcal{A}, F, \Delta)$, where $Q = \{1, 2, 3\}$, $\mathcal{A} = \{a2, a1, a0\}$, $F = \{3\}$, and Δ contains these transition rules:

$$\begin{aligned} a0 &\to 1 \\ a1(1) &\to 1 \\ a2(1,1) &\to 1 \\ a1(1) &\to 2 \\ a2(1,2) &\to 3 \end{aligned}$$

The automaton $M_{ta2}(p_{2r})$ is depicted in Figure 3.8. △

3.4 Computing subtree repetitions

Enumerating all subtree repeats and fast identification of the same subtrees within a tree is important, therefore many algorithms were designed to compute all subtree repeats. Two

Table 3.1: Subtree repeats table for the prefix notation $SRT\_post(t_{4r})$ of tree t_{4r}.

node id	1	2	3	4	5	6	7
$post(t_{4r})$	$a2$	$a2$	$a0$	$a0$	$a2$	$a0$	$a0$
repeat id	2	1	0	0	1	0	0

notable approaches shall be presented. The first is based on automata approach, and the second is based on dynamic programming.

The representation of repeats for use in this thesis was chosen to be close to the linear representation of the tree itself, which is different from the representation used by both presented state-of-the-art algorithms.

Definition 3.4.1. Let t and $pref(t)$ be a tree and its prefix notation, respectively. A *subtree repeats table* for $pref(t)$ denoted by $SRT\_pref(t)$ is an array of integers, where non-repeated (unique) subtree of t rooted on position i in the prefix notation $pref(t)$ is represented by unique integer value on position i and repeated subtrees on positions j, k, ... are represented by the same but otherwise unique integer value on positions j, k, ... in the $SRT\_pref(t)$.

Example 3.4.2. Consider a tree t_{4r} in the prefix notation $pref\ (t_{4r}) = a2\ a2\ a0\ a0\ a2\ a0\ a0$ over alphabet $\mathcal{A} = \{a3, a2, a1,\ a0\}$. Table 3.1 shows the $SRT\_pref(t_{4r})$ constructed for the tree t_{4r}. $\triangle$

Similar definitions for linear notations of trees other than the prefix notation are a straightforward modification of the Definition 3.4.1.

3.4.1 Automata approach to compute all subtree repeats

Subtree pushdown automata can be used to find all repetitions of subtrees. The approach using the postfix bar notation was presented in [30], however, similar result can be obtained using any other linear notation. The algorithm to compute all subtrees of a given tree requires a construction of the deterministic subtree pushdown automaton which was already discussed in Section 3.1.

The algorithm follows traces of accepting all subtrees with the subtree pushdown automaton. When the pushdown automaton would accept a subtree, the state, where the trace of transitions ended is investigated. States with non-singleton d-subset represent repetition of some subtree and a state with singleton d-subset represent subtree that is unique inside the processed tree. The original paper focuses only on repeated subtrees however both the repeats of subtrees and unique subtrees are important.

The resulting structure of subtree repeats as proposed in [30] is presented in form of a table with a row for each repeating subtree. Each row of the table contains positions where the subtree ends, representation of the subtree in postfix bar notation, and the type of the repetition. Listed types can represent F for the first occurrence, S for a square repetition, and G for a repetition with a gap between previous and the next subtree repeat.

To retrieve all subtree repetitions and all unique subtrees of a tree t in prefix notation $pref(t)$ from the $M_{dps}(pref(t))$ in a form of a table $SRT\_pref(t)$ as proposed in Definition 3.4.1, one can use the Algorithm 7 (RepeatsFromSubtreePushdownAutomaton) which is a variation on the algorithm presented in [30].

```
Name: RepeatsFromSubtreePushdownAutomaton
Input: Subtree pushdown automaton $M_{dps}(pref(t)) = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ for
       tree $pref(t)$ of size $n$
Output: Array $SRT\_pref(t)$ initialised to array of size $n$ representing the result
Input: $state$ the processed state initialised to $q_0$
Input: $size$ the length of linear representation of the subtree repeat initialised to 0
Input: $ac$ the arity checksum initialised to 1
1  begin
2      if $ac = 0$ then
3          let $subset = r_1, r_2, \ldots, r_{|state|}$ such that $r_i > r_{r-1}$;
4          foreach $r_i \in subset$ do
5              $SRT\_pref(t)[r_i - size] := r_0 - size$;
6          end
7      end
8      else
9          foreach $a_i \in \mathcal{A}$ do
10             for $(next\_state, S^{Arity(a_i)})) \in \delta(state, a_i, S)$ in transitions of automaton
                $M_{dps}(pref(t))$ do
11                 RepeatsFromSubtreePushdownAutomaton($M_{dps}(pref(t))$,
                    $SRT\_pref(t)$, $next\_state$, $size + 1$, $ac - 1 + Arity(a_i)$);
12             end
13         end
14     end
15 end
```

Algorithm 7: Algorithm of subtree repeats table construction from subtree pushdown automaton.

The algorithm is designed for the deterministic subtree pushdown automaton constructed for a tree in the prefix notation. Other notations can be used to construct the subtree pushdown automaton and later similar algorithm to the Algorithm 7 (RepeatsFromSubtreePushdownAutomaton) can be designed as well.

Given the subtree pushdown automaton $M_{dps}(pref(t))$, the running time to obtain the subtree repeats table $SRT\_pref(t)$ is linear with the sum of sizes of all subtrees of the tree t since it traces all transitions accepting subtrees of the tree t in the automaton symbol at a time.

3.4.2 Dynamic programming approach to compute all subtree repeats

A dynamic programming approach to find all subtree repeats was presented in [11]. The algorithm is designed to work on the tree in its postfix notation $post(t)$ of size n.

The algorithm internally processes triplets consisting of parts named S, l, ac. These triplets represent the same factors of the postfix representation of the tree $post(t)$. Starting positions of the factors are stored in the triplet part called S, the length of the factor is named l and the arity checksum value of the factor is named ac. Intuitively, the only interesting triplets are those with arity checksum ac equal to 0, they represent subtrees of the original tree. The algorithm eventually outputs these triplets. Each triplet (S, lac) that represents a subtree repeat is outputted exactly once, which allows aggregating the subtree repeats by an array $SRT\_post(t)$. The $SRT\_post(t)[i] = S[0] + l$, for all $i \in S$.

To use the information about repeats with some other linear notation, like prefix, or bar notation, the same array of repeats needs to be obtained. The result of the aforementioned algorithm can be transformed to correspond to the prefix notation by the following algorithm.

```
Name: PostfixToPrefixRepeats
Input: A tree T in postfix notation $post(t)$ of length $n$
Input: Array of subtree repeats $SRT\_post(t)$ constructed for postfix notation
Output: Array $SRT\_pref(t)$ initialised to array of size $n$ representing the result
Input/Output: $read\_position$ is an index to the $post(t)$ initialised to $n$
Input/Output: $write\_position$ is an index to the $post(t)$ initialised to $n$
1 begin
2     $rootIndex := read\_position$;
3     $read\_position := read\_position - 1$;
4     for $i := 1$ to $Arity(pref(T)[rootIndex])$ do
5         PostfixToPrefixRepeats ( $post(T)$, $SRT\_post(T)$, $SRT\_pref(T)$,
            $read\_position$, $write\_position$ );
6     end
7     $SRT\_pref(T)[write\_position] := SRT\_post(T)\ [\ rootIndex\ ]$;
8     $write\_position := write\_position - 1$;
9 end
```

Algorithm 8: Transformation algorithm of subtree repeats array from postfix to prefix.

Theorem 3.4.3. Given a tree t of size n, the algorithm to compute all subtree repeats presented in [11] with its result post-processed by Algorithm 8 (PostfixToPrefixRepeats) correctly produces the subtree repeats table for the prefix notation $SRT\_pref(t)$.

Proof. The algorithm to compute all subtree repeats presented in [11] is proven to be correct in [11]. Capturing its result in an array representing $SRT\_post(t)$ is straightforward processing of its input.

Given a tree consisting of only a root node, hence node labelled by a nullary symbol, the algorithm does not recurse and the node is simply copied to the output.

Given a tree rooted by a node labelled by non-nullary symbol, the algorithm first recursively processes the sequence of the children of the root in the right to left order, while it remembers the results in a result array filled from right to left. Finally, the root node is outputted to the resulting array.

The algorithm therefore correctly transforms the *SRT_post* to *SRT_pref*. □

Theorem 3.4.4. Given a tree t of size n, the algorithm to compute all subtree repeats presented in [11] with its result post-processed by Algorithm 8 (PostfixToPrefixRepeats) runs in time linear with respect to the size of the tree t.

Proof. The running time of the algorithm to compute all subtree repeats is shown to be linear with respect to the size of the processed tree in [11].

The transformation from $SRT\_post(t)$ to $SRT\_pref(t)$ constructed for a tree t of size n is linear with the size of the tree t since it reads from each position in the $SRT\_post(t)$ and writes to each position of the $SRT\_pref(t)$ exactly once and both the $SRT\_post(t)$ and the $SRT\_pref(t)$ are of size n. □

A similar algorithm of a transformation of *SRT_post* to *SRT_pref_bar* is presented as well. Other transformation algorithms are not shown.

```
Name: PostfixToPrefixBarRepeats
Input: A tree T in postfix notation $post(t)$ of length $n$
Input: Array of subtree repeats $SRT\_post(t)$ constructed for postfix notation
Output: Array $SRT\_pref\_bar(t)$ initialised to array of size $2n$ representing the
        result
Input/Output: $read\_position$ is an index to the $post(t)$ initialised to $n$
Input/Output: $write\_position$ is an index to the $pref\_bar(t)$ initialised to $2n$
1  begin
2      $rootIndex := read\_position$;
3      $read\_position := read\_position$ - 1;
4      $SRT\_pref\_bar(T)[write\_position] := \uparrow$;
5      $write\_position := write\_position$ - 1;
6      for $i := 1$ to $Arity(pref(T)[rootIndex])$ do
7          PostfixToPrefixRepeats ( $post(T)$, $SRT\_post(T)$, $SRT\_pref\_bar(T)$,
             $read\_position$, $write\_position$ );
8      end
9      $SRT\_pref\_bar(T)[write\_position] := SRT\_post(T)$ [ $rootIndex$ ];
10     $write\_position := write\_position$ - 1;
11 end
```
Algorithm 9: Transformation algorithm of subtree repeats array from postfix to prefix.

Theorem 3.4.5. Given a tree t of size n, the algorithm to compute all subtree repeats presented in [11] with its result post-processed by Algorithm 9 (PostfixToPrefixBarRepeats) correctly produces the subtree repeats table for the prefix notation $SRT\_pref\_bar(t)$.

Proof. The proof is similar to the proof of Theorem 3.4.3. The only difference in the Algorithm 9 (PostfixToPrefixBarRepeats) compared to Algorithm 8 (PostfixToPrefixRepeats) is an additional output of the bar symbol $\uparrow$. The order of non-bar symbols in the output $SRT\_pref\_bar(t)$ is the same as in the $SRT\_pref(t)$. Therefore only the correctness of positions of $\uparrow$ symbol in the output is needed to prove.

The bar symbol $\uparrow$ is always outputted first of the symbols forming a subtree of the tree, i.e. on the position with the highest index of the symbols forming the tree. That is in contrast with the non-bar symbols that are outputted last, i.e. with the lowest index of the symbols forming the tree. Therefore the non-bar symbol and the bar symbol are together forming the left and right parenthesis wrapping exactly the representations of the child nodes.

The algorithm therefore correctly transforms the SRT_post to SRT_pref_bar. ☐

Theorem 3.4.6. Given a tree t of size n, the algorithm to compute all subtree repeats presented in [11] with its result post-processed by Algorithm 9 (PostfixToPrefixBarRepeats) runs in time linear with respect to the size of the tree t.

Proof. The proof is similar to the proof of Theorem 3.4.4. Again, the running time of the algorithm to compute all subtree repeats is shown to be linear with respect to the size of the processed tree in [11].

The transformation from $SRT\_post(t)$ to $SRT\_pref\_bar(t)$ constructed for a tree t of size n is linear with the size of the tree t since it reads from each position in the $SRT\_post(t)$ and writes to each position of the $SRT\_pref\_bar(t)$ exactly once and the $SRT\_post(t)$ is of size n and the $SRT\_pref\_bar(t)$ is of size $2n$. ☐

Chapter **4**

Main Results in Tree Indexing

Tree indexing is covered by two approaches both covered in this chapter. The chapter summary and experimental results is presented to conclude the tree indexing topic of this thesis.

4.1 Basic nonlinear tree pattern pushdown automaton

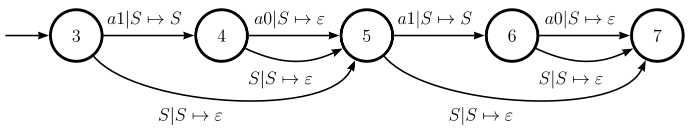
This section contains base results from an individual work presented as a conference paper [62] and as well as a journal paper [60].

Definition 4.1.1. Let t and $pref(t)$ be a tree and its prefix notation, respectively. A *nonlinear tree pattern pushdown automaton* for $pref(t)$ accepts all nonlinear tree patterns in prefix notation which has at most one nonlinear variable and match the tree t.

The nonlinear tree pattern pushdown automaton is based on the tree pattern pushdown automaton. Copies of parts of the tree pattern pushdown automaton called tails are added to construct the nonlinear tree pattern pushdown automaton from the tree pattern pushdown automaton. These tails represent parts of the automaton after reading a subtree wildcard.

Definition 4.1.2. Given a tree pattern pushdown automaton $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ and a state $q_t \in Q$, the $tail(M, q_t) = (Q_t, \mathcal{A}, G, \delta_t, q_t, S, F)$. $Q_t = Q \setminus Q_{us}$, Q_{us} is a set of unreachable states from q_t, $\delta_t = \delta \setminus \delta_{us}$, δ_{us} are transitions leading from or to state $q_n \in Q_{us}$.

Example 4.1.3. Consider a tree t_{1r} from Example 2.2.1 and its index, a tree pattern pushdown automaton $M_{npt}(t_{1r})$ from Example 3.2.5, which is depicted in Figure 3.5. The tail of automaton $M_{npt}(t_{1r})$ with initial state $q_t = 3$ is $tail(M_{npt}(t_{1r}), q_t) = (\{3, 4, 5, 6, 7\}, \mathcal{A} \cup \{S\}, \{S\}, \delta, 3, S, \varnothing)$. The corresponding transition diagram is illustrated in Figure 4.1. △



$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
3
4
5
6
7

Figure 4.1: A tail $tail(M_{npt}(t_{1r}), 3)$ of the tree pattern pushdown automaton $M_{npt}(t_{1r})$ from Example 4.1.3.

We note that every node of a tree t is a root of just one subtree, which is represented by symbol S. The prefix notation of such subtree is a factor of $pref(t_{1r})$. These factors are in the tree pushdown automaton "skipped" by transitions for wildcard symbol S.

Definition 4.1.4. Given a tree pattern pushdown automaton $M_{npt}(t) = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ and a state $q \in Q$, the *subtree skipped by transition* $sst(q) = b_1 b_2 \ldots b_m$, where $b_1, b_2, \ldots, b_m \in \mathcal{A} \setminus \{S\}$, is given by a labelled path $b_1, b_2, \ldots, b_m$ in the PDA $M_{npt}(t)$ between states q and q_t, where $(q_t, \varepsilon) \in \delta(q, S, S)$.

Informally, the $sst(q)$ is the prefix notation of the subtree which is skipped by transition reading S leading from the state q. The $sst(q)$ is used in Algorithm 10 (ConstructTail) to determine which subtree of the subject tree was "assigned" to a particular automaton tail.

Example 4.1.5. Consider a tree t_{1r} from Example 2.2.1 and a tree pattern pushdown automaton $M_{npt}(t_{1r})$ from Example 3.2.5, which is an index of t_{1r}. The subtree skipped by transition $sst(1) = a2\ a0\ a1\ a0$. △

The construction of basic nonlinear tree pattern PDA consists of two algorithms. Algorithm 10 (ConstructTail) constructs tails from the original tree pattern pushdown automaton. Algorithm 11 (ConstructBasicNTPPDA) recursively connects these created tails to the pushdown automaton being created.

The Algorithm 10 (ConstructTail) is similar to the Algorithm 11 (ConstructBasicNTP-PDA). The difference between them is that Algorithm 10 (ConstructTail) calls itself only when processing transition for symbol S leading from state q, where the $sst(q)$ equals its subtree parameter. On the other hand, Algorithm 11 (ConstructBasicNTPPDA) calls Algorithm 10 (ConstructTail) for each transition for symbol S.

Example 4.1.6. Given a string $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is a prefix notation of tree t_{1r} from Example 2.2.1, the corresponding basic nondeterministic nonlinear tree pattern pushdown automaton is $M_b(t_{1r}) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \emptyset)$, where its transition diagram is illustrated in Figure 4.2.

The basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t_{1r})$ consist of eleven tails and one original tree pattern pushdown automaton. To distinguish the tails, the states forming it are indexed by 1 to 11.

Tails with states indexed by numbers 1, 3, 4, 7, 8, 9, and 10 are the tails, where nonlinear variable X stands for a subtree represented in prefix notation as $a0$. Tails with states indexed by numbers 1, 4, and 8 represent the tails in which the nonlinear variable X

```
   Name: ConstructTail
   Input: Tail of nondeterministic tree pattern pushdown automaton $M_{tnpt}$
   Input: String representing subtree skipped by transition $x$
   Result: Recursively created tail $(M_{tnpt}, x)$
1  begin
2      foreach transition $(q_t, \varepsilon) \in \delta(q, S, S)$ in automaton $M_{tnpt}$ where the $sst(q) = x$
        do
3          Create $(M_{tmp}, x) = ConstructTail(tail(M_{tnpt}, q_t), x)$ using Algorithm 10
            (ConstructTail);
4          Add new state $q_{id}$ to $M_{tnpt}$ where $q_{id}$ is copy of state $q_t$;
5          Add new transition $(q_{id}, \varepsilon) \in \delta(q, X, S)$ to $M_{tnpt}$;
6          Add $M_{tmp}$ to $M_{tnpt}$ and merge initial state of $M_{tmp}$ with $q_{id}$;
7      end
8      return $(M_{tnpt}, x)$;
9  end
```

Algorithm 10: Recursive construction of tail of nondeterministic basic nonlinear tree pattern automaton.

```
   Name: ConstructBasicNTPPDA
   Input: Nondeterministic tree pattern pushdown automaton $M_{npt}(t)$
   Result: Basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$
1  begin
2      foreach transition $(q_t, \varepsilon) \in \delta(q, S, S)$ in automaton $M_{npt}(t)$ do
3          Create $(M_{tmp}, sst(q)) = ConstructTail(tail(M_{npt}(t), q_t), sst(q))$ using
            Algorithm 10 (ConstructTail);
4          Add new state $q_{id}$ to $M_{npt}(t)$ where $q_{id}$ is copy of state $q_t$;
5          Add new transition $(q_{id}, \varepsilon) \in \delta(q, X, S)$ to $M_{npt}(t)$;
6          Add $M_{tmp}$ to $M_{npt}(t)$ and merge initial state of $M_{tmp}$ with $q_{id}$;
7      end
8      $M_b(t) = M_{npt}(t)$;
9      return $M_b(t)$;
10 end
```

Algorithm 11: Construction of basic nondeterministic nonlinear tree pattern pushdown automaton.

is read once. Tails with states indexed by numbers 3, 7, and 10 represent the tails in which the nonlinear variable X is read twice. A tail with state indexed by number 9 represents the tail in which the nonlinear variable X is read three times.

Tails with states indexed by numbers 2, 5, and 6 are the tails, where nonlinear variable X stands for subtree represented in prefix notation as $a1\ a0$. Tails with states indexed by numbers 2 and 6 represent the tails in which the nonlinear variable X is read once. A tail with state indexed by number 5 represents the tail in which the nonlinear variable X is read twice.

A tail with state indexed by numbers 11 is the tail in which the nonlinear variable X is read once and where nonlinear variable X stands for subtree represented in prefix notation as $a2\ a0\ a1\ a0$.

The corresponding basic deterministic nonlinear tree pattern pushdown automaton is $M_{db}(t_{1r}) = (Q, \mathcal{A} \cup \{S, X\}, \delta, 0, \varnothing)$, where its transition diagram is illustrated in Figure 4.3.

$\triangle$

4.2 Nonlinear tree pattern pushdown automaton

Some states of the basic nondeterministic nonlinear tree pattern pushdown automaton constructed by Algorithm 11 (ConstructBasicNTPPDA) can be merged so that states in nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t) = (Q,\ \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0,\ S,\ \varnothing)$ for a subject tree t would still track both assigned subtree and the same number of nonlinear variables read from the pattern. Merged states are those from tails with the same assigned subtree, and the same number of nonlinear variables read.

Definition 4.2.1. Let $M_b(t) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, q_0, S, \varnothing)$ be a basic nondeterministic nonlinear tree pattern PDA constructed by Algorithm 11 (ConstructBasicNTPPDA). Let $q \in Q$ and x be the longest string over alphabet $\mathcal{A} \setminus \{S, X\}$, where $(q, x, \alpha) \vdash^*_{Mb} (q_f, \varepsilon, \beta)$. The *tree node state label* $tnsl(q)$ is defined $tnsl(q) = |pref(t)| - |x|$.

Informally, the $tnsl(q)$ is a function that erases the index from the state label q in sample automata tails.

Example 4.2.2. Given a basic nondeterministic nonlinear tree pattern pushdown automaton is $M_b(t_{1r}) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \varnothing)$, its transition diagram is shown in Figure 4.2.

Then, $tnsl(3) = 3$, $tnsl(5_4) = 5$, $tnsl(7_{11}) = 7$, $tnsl(7_9) = 7$. $\triangle$

Definition 4.2.3. Given a basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta,\ 0, S, \varnothing)$ created by Algorithm 11 (ConstructBasicNTPPDA), the *number of nonlinear variable transitions* $nnv(q, X)$ is the number of transitions reading nonlinear variable X on the path from the initial state q_0 to state q, where q and $q_0 \in Q$.

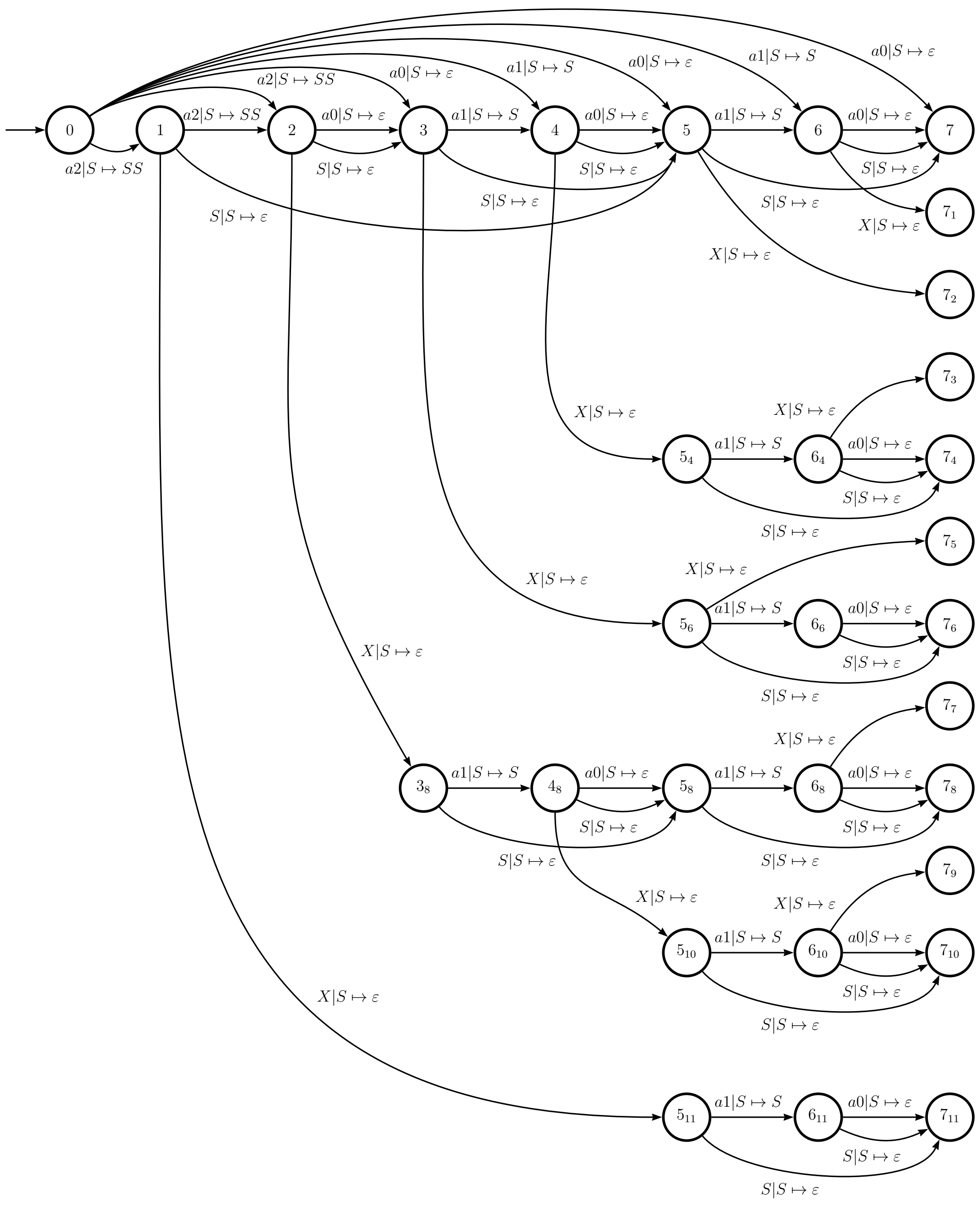


Figure 4.2: Basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t_{1r})$ from Example 4.1.6 constructed for tree t_{1r} shown in Figure 2.1a.

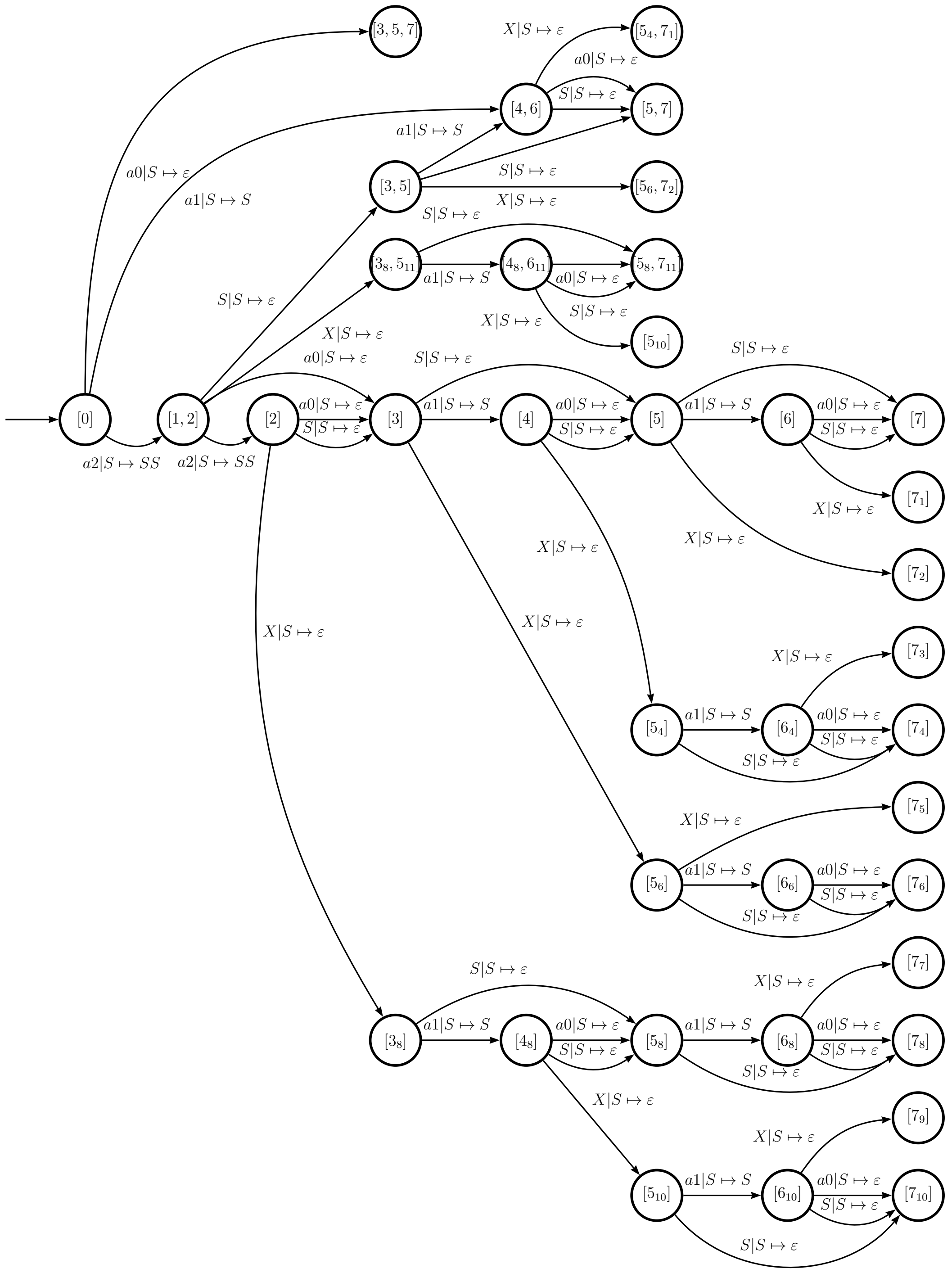


Figure 4.3: Basic deterministic nonlinear tree pattern pushdown automaton $M_{db}(t_{1r})$ from Example 4.1.6 constructed for subject tree t_{1r} shown in Figure 2.1a.

```
Name: ComputeTreeNodeStateLabel
Input: Basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$
Input: State $q$ for which the tnsl is counted
Result: Number representing tnsl
begin
    $n = 0$;
    initial is the starting state of $M_b(t)$;
    loop
        if exists transition $(q, S^{arity(a)}) \in \delta(q_{prev}, a, S)$ where $a \in \mathcal{A} \setminus \{S, X\}$ and
          $q_{pref} \neq initial$ then
            $n = n + 1$, $q = q_{prev}$;
        else if exists transition $(q, \varepsilon) \in \delta(q_{prev}, X, S)$ where $X$ is nonlinear variable
          then
            $n = n + |tnst(q_{prev})|$, $q = q_{prev}$;
        else
            return $n + 1$;
        end
    end
end
```

Algorithm 12: Algorithm for counting the *tnsl*.

```
Name: ComputeTheNumberOfNonlinearVariables
Input: Basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$
Input: State $q$ for which the nnv is counted
Result: Number representing nnv
begin
    $n = 0$;
    initial is the starting state of $M_b(t)$;
    loop
        if exists transition $(q, S^{arity(a)}) \in \delta(q_{prev}, a, S)$ where $a \in \mathcal{A}$ an
          $q_{pref} \neq initial$ then
            $q = q_{prev}$;
        else if exists transition $(q, \varepsilon) \in \delta(q_{prev}, X, S)$ where $X$ is nonlinear variable
          then
            $n = n + 1$;
            $q = q_{prev}$;
        else
            return $n$;
        end
    end
end
```

Algorithm 13: Algorithm for counting the *nnv*.

Definition 4.2.4. Given a state r, where $nnv(r) \geq 1$ of the basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$, that state r is inside the automaton in some tail. The *denoting state* of the tail containing r the $tds(r)$ is state s of the automaton $M_b(t)$ that satisfies $(s, X\Omega, S\Gamma) \vdash^*_{Mb(t)} (r, \Omega, \Gamma)$, where $\Omega = \mathcal{A} \setminus \{S, X\}$ and $\Gamma \in S*$.

Definition 4.2.5. Given a basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$ created by Algorithm 11 (ConstructBasicNTPPDA), the *mergeable states* $ms(M_b(t))$ is a mapping of tuple ($tnst$, nnv, $tnsl$) to set of states mss. The set mss is a set of states $\{s_1, s_2, \dots : nnv(s_1, X) = nnv(s_2, X)$ and $tnst(tds(s_1)) = tnst(tds(s_2))$ and $tnsl(s_1) = tnsl(s_2); s_1, s_2 \in Q\}$.

Informally, the states q of mss have the same number of transitions reading nonlinear variable X on the path from the initial state of the automaton to them, are part of a tail assigned with the same subtree, and have the same tree node state label.

Each set from the collection of sets of mergeable states $ms(M_b(t))$ defines states of basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$ that can be merged. When the states are merged, the resulting automaton is called nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t)$.

Example 4.2.6. Given a string $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is the prefix notation of tree t_{1r} from Example 2.2.1. The corresponding basic nondeterministic nonlinear tree pattern pushdown automaton is $M_b(t_{1r}) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \varnothing)$, where its transition diagram and states are illustrated in Figure 4.2.

$$ms(M_b(t_{1r})) = \left\{ \begin{array}{l} (a0, 1, 5) \mapsto \{5_4, 5_8\}, (a0, 1, 6) \mapsto \{6_4, 6_8\}, (a0, 1, 7) \mapsto \{7_1, 7_4, 7_8\}, \\ (a0, 2, 7) \mapsto \{7_3, 7_7, 7_{10}\}, (a1\ a0, 1, 7) \mapsto \{7_2, 7_6\}, \dots \end{array} \right\}$$

The entries of $ms(M_b(t_{1r}))$ where the size of the mapped set mss is 1 are omitted. △

Example 4.2.7. Given a string $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is the prefix notation of tree t_{1r} from Example 2.2.1, the corresponding nondeterministic nonlinear tree pattern pushdown automaton is $M_{nnpt}(t_{1r}) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \varnothing)$, where merged states are in Example 4.2.6 and its transition diagram and states are illustrated in Figure 4.4.

Deterministic version of the pushdown automaton the deterministic nonlinear tree pattern pushdown automaton $M_{dnpt}(t_{1r})$ created from automaton $M_{nnpt}(t_{1r})$ by standard determinisation is shown in Figure 4.5. △

4.2.1 Time and space complexity analysis

Lemma 4.2.8. The time complexity of accepting the nonlinear tree template by automaton created by Algorithm 14 (ConstructNTPPDA) is $\mathcal{O}(\sum_K k_i)$, where K is the set of all prefixes except ε, and k_i is the number of distinct sequences of transitions in automaton $M_{nnpt}(t)$ for $k_i \in K$ which ends in a state of automaton M_{nnpt}.

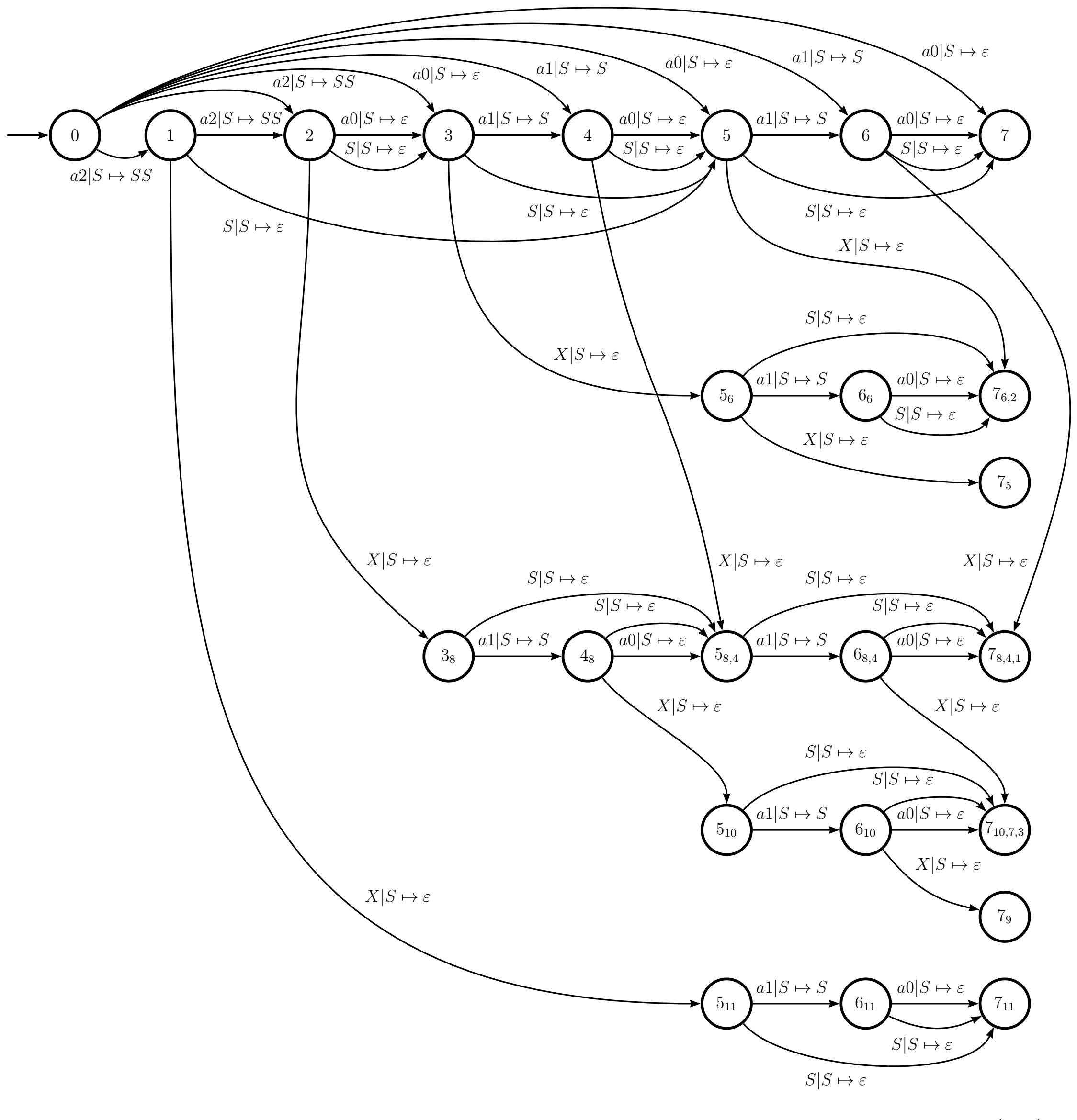


Figure 4.4: Nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t_{1r})$ from Example 4.2.7 constructed by Algorithm 14 (ConstructNTPPDA) for subject tree t_{1r} shown in Figure 2.1a.

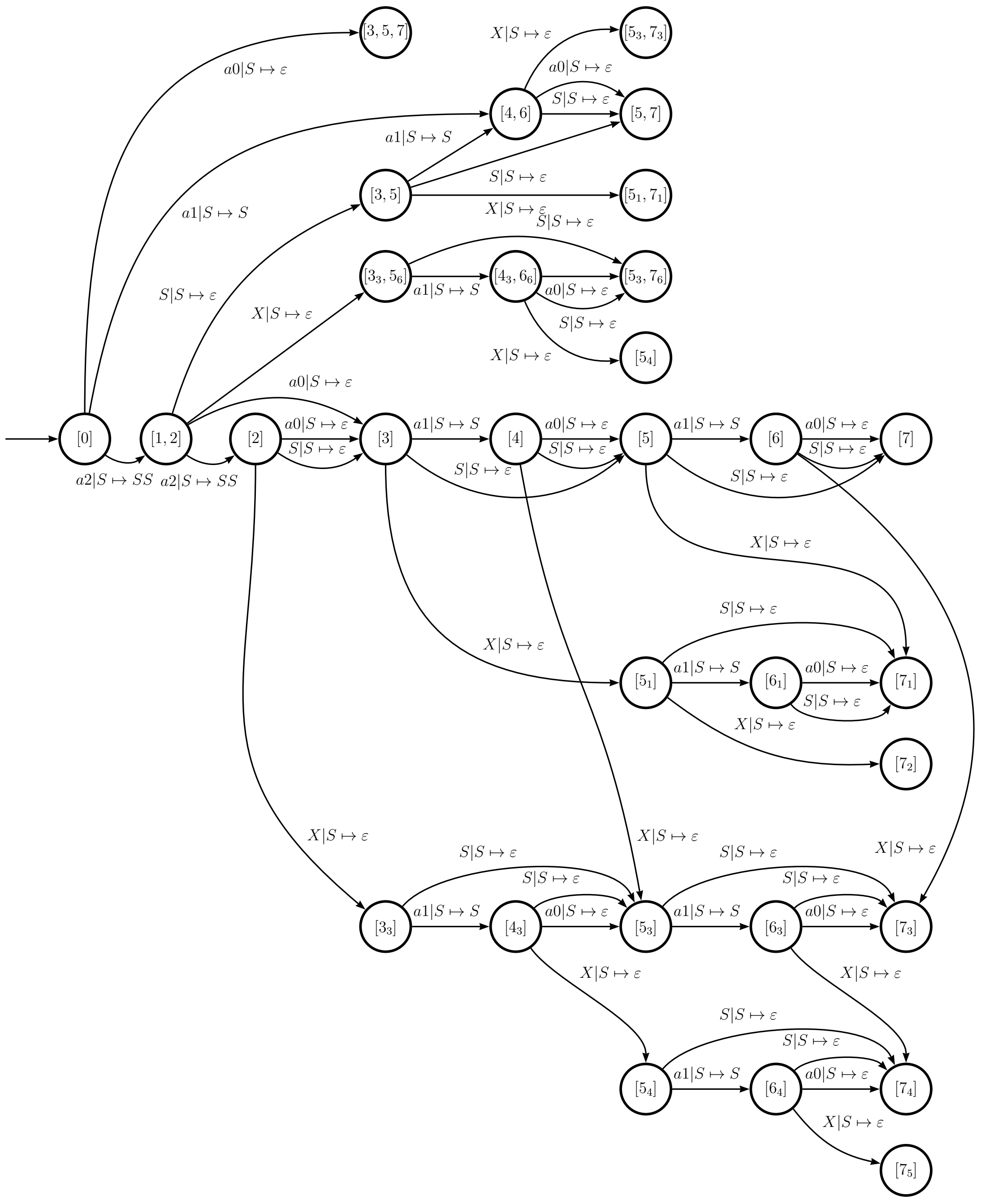


Figure 4.5: Deterministic nonlinear tree pattern pushdown automaton $M_{dnpt}(t_{1r})$ from Example 4.2.7 constructed by Algorithm 14 (ConstructNTPPDA) for subject tree t_{1r} shown in Figure 2.1a.

```
Name: ConstructNTPPDA
Input: Basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$
Result: Nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t)$
begin
    foreach transitions $(s, S^{(Arity(a))}) \in \delta(q, a, S)$ do
        if $nnv(s, X) \neq 0$ then
            if the key $(tnst(tds(s)), nnv(s, X), tnsl(s))$ is undefined in mapping $ms(M_b(t))$ then
                create an empty set on that key;
            end
            Add $s$ to the collection $ms(M_b(t))$ to the set on the key $(tnst(tds(s)), nnv(s, X), tnsl(s))$;
        end
    end
    foreach mapped set in the collection $ms(M_b(t))$ do
        Merge all states in this set in state $M_b(t)$ to produce $M_{nnpt}(t)$;
    end
    return $M_{nnpt}(t)$;
end
```

Algorithm 14: Construction of the nondeterministic nonlinear tree pattern pushdown automaton.

Proof. Automata have to try all possible sequences of transitions according to tree template which occurs in the nondeterministic nonlinear tree pattern automaton. Sequences of symbols of these transitions form a prefix of tree template. The prefix of the size of one symbol from tree template is handled by exactly n steps, where n is the number of all possible sequences of transitions in the automaton for that prefix. Prefix of the size of two symbols is handled by $n + m$ steps, where m is the number of all possible sequences of transitions in the automaton for that prefix. Note that handling two symbols prefix requires two transitions to be processed, however the first transition is already accounted by the prefix of a size of one symbol.

Exact time complexity is then the sum of all possible sequences of transitions in the automaton for all prefixes of nonlinear tree template, which is $\mathcal{O}(\sum_S rs_i)$. □

Lemma 4.2.9. The number of states of nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t)$ created by Algorithm 14 (ConstructNTPPDA) is $\mathcal{O}(n(\sum_{i=0}^{s} r_i)) = \mathcal{O}(n^2)$, where n is the number of nodes of the subject tree, s is the number of distinct subtrees, and r_i is the number of repetitions of each subtree.

Proof. An occurrence of each unique subtree in a tree increments the number of automaton tails, which are created for this subtree. The exact number of tails created for particular subtree is then r_i, where r_i is the number of repetitions of that subtree. Then the total number of tails for one nonlinear variable in automaton is the number of tails created for each unique subtree of the indexed tree which is $\sum_{i=0}^{s} r_i$.

The total number of tails does not count the original automaton. The exact number of states of the automaton for one nonlinear variable is $\mathcal{O}(n(\sum_{i=0}^{s} r_i+1)) = \mathcal{O}(n(\sum_{i=0}^{s} r_i))$. □

Lemma 4.2.10. The number of transitions of nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t)$ created by Algorithm 14 (ConstructNTPPDA) is $\mathcal{O}(n^2 + n+\sum_{i=0}^{s}(\frac{r_i^2+r_i}{2})) = \mathcal{O}(n^2)$, where n is the number of nodes of a subject tree, s is the number of distinct subtrees and r_i is the number of repetitions of each unique subtree.

Proof. For all tails used to construct the nondeterministic nonlinear tree pattern PDA, there are transitions reading nonlinear variable X between these tails. There is one transition heading to the last tail. There are two transitions heading to the previous tail, and so on. The number of transitions reading nonlinear variable X is $\sum_{i=0}^{s}(\frac{r_i^2+r_i}{2})$.

Using Lemma 4.2.9 the number of transitions for symbol S is $\mathcal{O}(n^2)$ as all states are a source of a subtree skipping transition.

The number of transitions for symbol $a \in \mathcal{A}$ is $\mathcal{O}(n^2 + n)$ as all states are a source of a tree node reading transition, except for the initial state which is a source of n tree node reading transitions.

The number of transitions then is $\mathcal{O}(n^2 + n + \sum_{i=0}^{s}(\frac{r_i^2+r_i}{2}))$. □

Lemma 4.2.11. Given a tree t with n nodes, the number of distinct nonlinear tree patterns which match the tree t can be at most $3^{n-1} + n - 1$.

Proof. First, subtrees of any subtree of the tree t can be replaced by wildcard S and the tree template resulting from such a replacement is a tree pattern which matches the tree.

Second, same subtrees of any subtree of the tree t can be replaced by nonlinear variable X and the nonlinear tree template resulting from such replacement is a nonlinear tree pattern which matches the tree.

Given a tree with n nodes, the maximal number of subsets of subtrees that can be replaced by wildcard S or nonlinear variable X occurs for the case of a tree t_{3r} whose structure is given by the prefix notation $pref(t_{3r}) = a(n-1)\ a0\ a0 \ldots a0$, where $n \geq 2$. Such a tree is illustrated in Figure 3.7. In this tree, each of the $n-1$ nullary symbols $a0$ can be replaced by the wildcard symbol S or nonlinear variable X, and therefore we can create 3^{n-1} distinct tree templates which are tree patterns matching the tree t_{3r}.

Third, the subtrees of the tree t are themselves valid nonlinear tree patterns. These give $n-1$ other distinct nonlinear tree patterns (provided that all subtrees are unique).

Thus, the total number of distinct tree patterns matching the tree t can be at most $3^{n-1} + n - 1$. □

4.3 Simple nonlinear tree pattern matching

The nondeterministic nonlinear tree pattern pushdown automaton can be even more minimised by omitting the $nnv(q, X)$ part of the key of the mapping $ms(M_b(t))$. The resulting automaton would represent an index of the subject tree for nonlinear tree pattern matching but would not be able to say how many nonlinear variables have been read during processing the nonlinear tree pattern.

The basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$ or nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t)$ is too complex for simple nonlinear tree pattern matching. The simple nondeterministic nonlinear tree pattern pushdown automaton $M_s(t)$ merges states that differ only by the $nnv(q, X)$, thus it is smaller, but it is unable to track the number of nonlinear variables. Merged states are those from tales with the same assigned subtree.

Definition 4.3.1. Given a basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$ created by Algorithm 11 (ConstructBasicNTPPDA), the *simple mergeable states* $sms(M_b(t))$ is a mapping of tuple ($tnst$, nnv, $tnsl$) to set of states $smss$. The set $smss$ is a set of states $\{s_1, s_2, \dots : tnst(tds(s_1)) = tnst(tds(s_2))$ and $tnsl(s_1) = tnsl(s_2); s_1, s_2 \in Q\}$.

Informally, the states q of $smss$ are part of a tail assigned with the same subtree, and have the same tree node state label.

Each set from the collection of sets of simple mergeable states $sms(M_b(t))$ defines states of basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$ that can be merged. When the states are merged, the resulting automaton is called simple nondeterministic nonlinear tree pattern pushdown automaton $M_s(t)$.

Example 4.3.2. Given a string $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is the prefix notation of tree t_{1r} from Example 2.2.1. The corresponding basic nondeterministic nonlinear tree pattern pushdown automaton is $M_b(t_{1r}) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \varnothing)$, where its transition diagram and states are illustrated in Figure 4.2.

$$sms(M_b(t_{1r})) = \left\{ \begin{array}{l} (a0, 5) \mapsto \{5_4, 5_8, 5_{10}\}, (a0, 6) \mapsto \{6_4, 6_8, 6_{10}\}, \\ (a0, 7) \mapsto \{7_1, 7_4, 7_8, 7_3, 7_7, 7_{10}, 7_9\}, (a1\ a0, 7) \mapsto \{7_2, 7_5, 7_6\}, \dots \end{array} \right\}$$

The entries of $sms(M_b(t_{1r}))$ where the size of the mapped set $smss$ is 1 are omitted. △

The Algorithm 15 (ConstructSimpleNTPPDA) is very similar to Algorithm 14 (ConstructNTPPDA). It also computes a mapping with set of states $smss$ but the key for each set consists only from $tnst$ and $tnsl$ fields. The nnv is omitted because the desired automaton will not track the number of nonlinear variables read. The checking for $nnv \neq 0$ of the target state of transitions is essential, the automaton would merge states that are from the original basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$ with states from tails handling the nonlinear variable.

Example 4.3.3. Given a string $pref(t_{1r}) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is a prefix notation of tree t_{1r} from Example 2.2.1, the corresponding simple nondeterministic nonlinear tree pattern pushdown automaton is $M_s(t_{1r}) = (Q, \mathcal{A} \cup \{S, X\}, \delta, 0, \varnothing)$, where its transition diagram and states are illustrated in Figure 4.6.

```
Name: ConstructSimpleNTPPDA
Input: Basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$
Result: Simple nondeterministic nonlinear tree pattern pushdown automaton
        $M_s(t)$
1  begin
2      foreach transitions $(s, S^{(Arity(a))}) \in \delta(q, a, S)$ do
3          if $nnv(s, X) \neq 0$ then
4              if the key $(tnst(tds(s)), tnsl(s))$ is undefined in mapping $sms(M_b(t))$
                 then
5                  create an empty set on that key;
6              end
7              Add $s$ to the collection $sms(M_b(t))$ to the set on the key $(tnst(tds(s)),$
                 $tnsl(s))$;
8          end
9      end
10     foreach mapped set in the collection $sms(M_b(t))$ do
11         Merge all states in this set in state $M_b(t)$ to produce $M_s(t)$;
12     end
13     return $M_s(t)$;
14 end
```
Algorithm 15: Construction of simple nondeterministic nonlinear tree pattern pushdown automaton.

Deterministic version of the pushdown automaton the simple deterministic nonlinear tree pattern pushdown automaton $M_{ds}(t_{1r})$ created from automaton $M_s(t_{1r})$ by standard determinisation is shown in Figure 4.7. △

4.3.1 Time and space complexity analysis

The time complexity of accepting a nonlinear tree pattern by the automaton $M_s(t)$ constructed for a subject tree t is the same as stated in Lemma 4.2.8.

Lemma 4.3.4. The number of states of simple nondeterministic nonlinear tree pattern pushdown automaton M_s (space complexity) created by Algorithm 15 (ConstructSimpleNTPPDA) is $\mathcal{O}(sn)$, where n is the number of nodes of a subject tree and s is the number of distinct subtrees.

Proof. The proof is constructed similarly to Lemma 4.2.9.

One tail is created for each unique subtree. Then the total number of tails for one nonlinear variable in pushdown automaton is the sum of tails created for each unique subtree of the indexed tree which is s the number of unique subtrees. The total number of tails does not count the original automaton. The exact space complexity of the simple

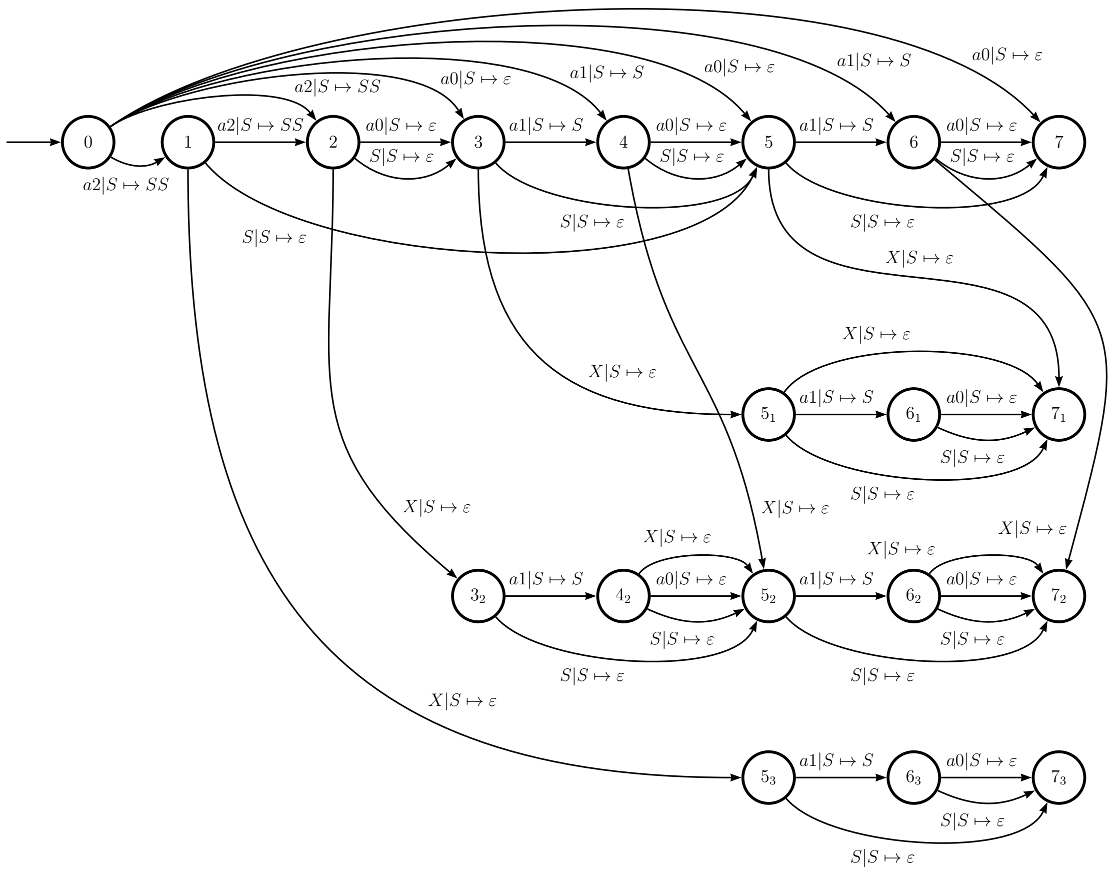


Figure 4.6: Simple nondeterministic nonlinear tree pattern pushdown automaton $M_s(t_{1r})$ from Example 4.3.3 constructed from Example subject tree t_{1r} shown in Figure 2.1a.

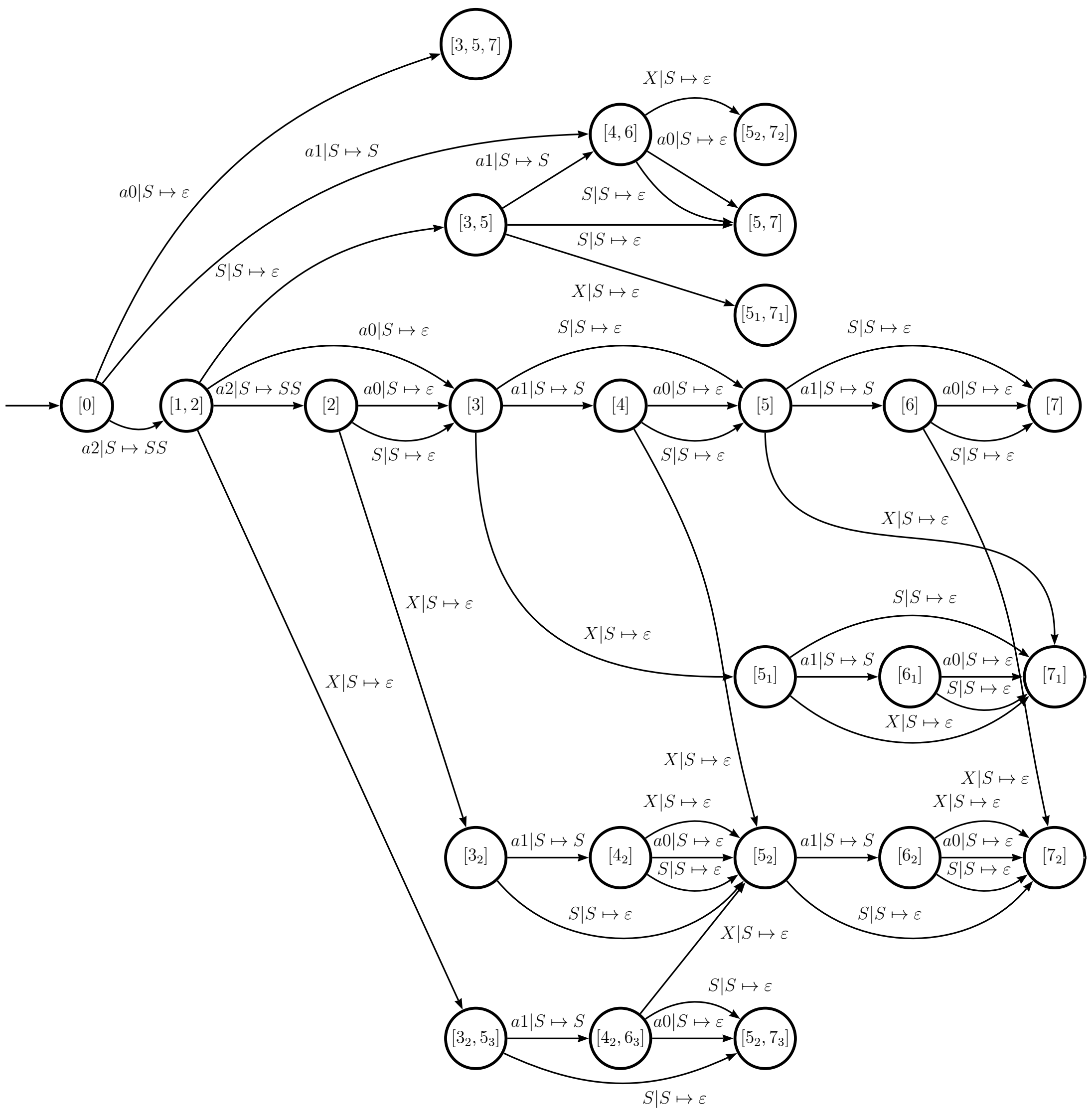
nondeterministic nonlinear tree pattern pushdown automaton for one nonlinear variable is $\mathcal{O}(sn)$.

Note that the complexity is also n, for no nonlinear variable, which means the original automaton and no other tails.

Note that if there is no repeating subtree in the subject tree, one new tail of original automaton is constructed for each node of the subject tree. The space complexity is therefore $\frac{1}{2}n^2 \in \mathcal{O}(n^2) = \mathcal{O}(ns)$. □

Lemma 4.3.5. The number of transitions of simple nondeterministic nonlinear tree pattern pushdown automaton M_s (space complexity) created by Algorithm 15 (ConstructSimpleNTPPDA) is $\mathcal{O}(\sum_{i=0}^{s}(r_i) + n^2)$, where n is the number of nodes of a subject tree, s is the number of distinct subtrees and r_i is the number of repetitions of each unique subtree.

Proof. The proof is constructed similarly to the proof of Lemma 4.2.10.



$[3,5,7]$
$X|S \mapsto \varepsilon$
$[4,6]$
$a0|S \mapsto \varepsilon$
$[5_2,7_2]$
$a1|S \mapsto S$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$[3,5]$
$[5,7]$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$[5_1,7_1]$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$[0]$
$[1,2]$
$a2|S \mapsto SS$
$[2]$
$a0|S \mapsto \varepsilon$
$[3]$
$a1|S \mapsto S$
$[4]$
$a0|S \mapsto \varepsilon$
$[5]$
$a1|S \mapsto S$
$[6]$
$a0|S \mapsto \varepsilon$
$[7]$
$a2|S \mapsto SS$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$[5_1]$
$a1|S \mapsto S$
$[6_1]$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$[7_1]$
$X|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$[3_2]$
$a1|S \mapsto S$
$[4_2]$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$[5_2]$
$a1|S \mapsto S$
$[6_2]$
$a0|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$[7_2]$
$S|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$X|S \mapsto \varepsilon$
$S|S \mapsto \varepsilon$
$[3_2,5_3]$
$a1|S \mapsto S$
$[4_2,6_3]$
$a0|S \mapsto \varepsilon$
$[5_2,7_3]$
$S|S \mapsto \varepsilon$

Figure 4.7: Simple deterministic nonlinear tree pattern pushdown automaton $M_{ds}(t_{1r})$ from Example 4.3.3 constructed from Example subject tree t_{1r} shown in Figure 2.1a.

Given the number of repetitions of each unique subtree of subject tree r_i the number of transitions for nonlinear variable X is $\sum_{i=0}^{s}(2r_i - 1)$.

Using Lemma 4.3.4 the number of transitions for symbol S is $\mathcal{O}(sn)$ as all states are a source of a subtree skipping transition.

The number of transitions for symbol $a \in \mathcal{A}$ is $\mathcal{O}(sn + n)$ as all states are a source of a tree node reading transition, except for the initial state which is a source of n tree node reading transitions.

The number of transitions then is $\mathcal{O}(\sum_{i=0}^{s}(2r_i - 1) + sn + n)$.

Note that there is at maximum three transitions for each state in basic nondeterministic nonlinear tree pattern pushdown automaton plus one transition for each state of the original nondeterministic tree pattern pushdown automaton. The number of transitions then is $3sn + n \in \mathcal{O}(sn) = \mathcal{O}(\sum_{i=0}^{s}(2r_i - 1) + sn + n)$. □

4.4 Processing more nonlinear variables in nonlinear tree patterns

This section contains an extension of results from the previous section, originally presented as an individual work in the journal paper [60].

Indexing for nonlinear tree pattern matching with more than one nonlinear variable can be done by a pushdown automaton created as a pushdown automaton for the intersection of languages. Automaton for two nonlinear variables would be constructed on the basis of two automata – each of them for one nonlinear variable. The disadvantage of this approach would be increasing space complexity.

Another approach is represented by a nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t)$ for one nonlinear variable that can be used as an indexing data structure also for nonlinear tree patterns with more variables. The idea is to compare which transitions of more runs of this single automaton were used to match the nonlinear tree pattern. The nonlinear tree pattern needs to be modified because it contains symbols representing the nonlinear variables that the nondeterministic nonlinear tree pattern pushdown automaton can't handle.

Example 4.4.1. Consider a ranked alphabet $\mathcal{A} = \{a4, a3, a2, a1, a0\}$. Consider a nonlinear tree template p_{7r} over $\mathcal{A} \cup \{S, Y, Z\}$ $p_{7r} = (\{a4_1,\ X_2,\ X_3,\ Y_4,\ Y_5\},\ R_{p4})$ over $\mathcal{A}$, where R_{p4} is a set of the following ordered pairs:

$$R_{p4} = \{(a4_1, X_2), (a4_1, X_3), (a4_1, Y_4), (a4_1, Y_5)\}.$$

Nonlinear tree template p_{7r} is illustrated in Figure 4.8.

Nonlinear tree template p_{7r} can be decomposed to nonlinear tree templates each for one of the nonlinear variables. These nonlinear tree templates will be over alphabet $\mathcal{A} \cup \{S, X\}$ and are illustrated in Figure 4.9. △

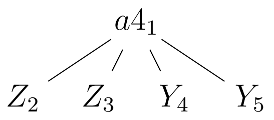


Figure 4.8: Nonlinear tree template p_{7r} from Example 4.4.1.

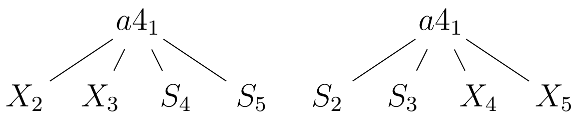


Figure 4.9: Decomposition of nonlinear tree template p_{7r} from Example 4.4.1.

In the beginning, the algorithm decomposes a given nonlinear tree template to nonlinear tree templates of one nonlinear variable. Then, the accepting sequences of transitions are computed using nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t)$ and each decomposed nonlinear tree pattern. These accepting sequences of transitions can be used for filtering real occurrences out of the original tree template with more nonlinear variables.

4.4.1 Time and Space Complexity Analysis

Lemma 4.4.2. Time complexity of accepting the nonlinear tree template with more nonlinear variables by nondeterministic nonlinear tree pattern pushdown automaton is $\mathcal{O}(v \times m + \sum run(pd) + \sum Occ_{pd})$, where v is the number of nonlinear variables, m is the size of the nonlinear template, $run(pd)$ is the time of locating all accepting transition sequences of each of decomposed nonlinear templates pd in automaton $M_{nnpt}(t)$ and Occ_{pd} is the size of occurrences of decomposed template pd. The $run(pd)$ is given by Lemma 4.2.8.

Proof. The nonlinear tree template needs to be decomposed to nonlinear tree templates for one nonlinear variable. This takes $v \times m$ time.

Occurrences of each nonlinear tree template from decomposed nonlinear tree template p are computed in time $\sum run(pd)$.

Composition of partial occurrences Occ_{pd} to Occ can be done in $\sum Occ_{pd}$ time. $\square$

Lemma 4.4.3. Given a tree t with n nodes, the number of distinct nonlinear tree patterns (with more nonlinear variables) which match the tree t can be at most $(2 + v)^{n-1} + n - 1$.

Proof. First, subtrees of any subtree of the tree t can be replaced by the wildcard symbol S and the tree template resulting from such a replacement is a tree pattern which matches the tree.

Second, same subtrees of any subtree of the tree t can be replaced by any, however same, nonlinear variable and the nonlinear tree template resulting from such replacement is a nonlinear tree pattern which matches the tree.

Given a tree with n nodes, the maximal number of subsets of subtrees that can be replaced by the wildcard or the nonlinear variables occurs for the case of a tree t_{3r} whose structure is given by the prefix notation $pref(t_{3r}) = a(n-1)\ a0\ a0 \ldots a0$, where $n \geq 2$.

```
Name: QueryNTPPDAByNonlinearTreePatternWithMoreVariables
Input: Nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t)$
Input: Nonlinear tree pattern with more variables $p$
Input: Set $vars$ of variables used in the template $p$
Result: Occurrences of the pattern
1  begin
2      Collection of nonlinear templates for one variable $po$ is an empty collection;
3      foreach variable $var$ in $vars$ do
4          $pd$ is a clone of nonlinear tree template $p$;
5          Change a symbol in leaf nodes to wildcard symbol $S$, where node label
             $l \in (vars \setminus var)$;
6          Change a symbol in leaf nodes to nonlinear variable $X$, where node label
             $l = var$;
7          Add $pd$ to $po$;
8      end
9      Set $Occ$ contains $\{0, 1, \ldots, n\}$ where $n$ is the size of the tree $t$;
10     foreach nonlinear tree template $pd$ in $po$ do
11         Determine accepting sequences of transitions $ts$ of tree template $t$ using
             $M_{nnpt}(t)$;
12         Compute a set $Occ_{pd}$ as set of $tnsl(q)$, where $q$ is a target state of the first
             transition from all $ts$;
13         Remove all items in $Occ$ which are not in $Occ_{pd}$;
14     end
15     return $Occ$;
16 end
```

Algorithm 16: Algorithm of nonlinear tree pattern matching with more nonlinear variables using nondeterministic nonlinear tree pattern pushdown automaton $M_{nnpt}(t)$.

Such a tree is illustrated in Figure 3.7. In this tree, each of the $n-1$ nullary symbols $a0, a0, \ldots, a0$ can be replaced by wildcard symbol S or any nonlinear variable and therefore we can create $(2+v)^{n-1}$ distinct tree templates which are tree patterns matching the tree t_{3r}.

Third, the subtrees of the tree t are themselves valid nonlinear tree patterns. These give $n-1$ other distinct nonlinear tree patterns (provided that all subtrees are unique).

Thus, the total number of distinct tree patterns matching the tree t can be at most $(2+v)^{n-1} + n - 1$. □

4.5 Linear space index of a tree for tree patterns

This section contains main results from joint work with Martin Poliak and Radomír Polách presented as a conference paper [40]. The original author of most parts of the algorithm is

Martin Poliak, however, the idea has been generalised in use of arbitrary indexing structure for strings. This section is included to summarise the results of tree indexing for tree patterns and present it for later extension.

4.5.1 Creation of an indexing structure

The section deals with the preprocessing phase, in which an index of a subject tree t is constructed. The index consists of two parts:

- An arbitrary indexing structure constructed for the $pref(t)$, by which occurrences of all substrings of $pref(t)$ can be located. See Chapter 2 for examples. We note that not all substrings of $pref(t)$ are subtrees in the prefix notation.
- A *subtree jump table for prefix notation*, a linear-size structure needed for finding positions of ends of subtrees represented by wildcard symbols S and nonlinear variables $X, Y, \dots$.

Definition 4.5.1. Let t and $pref(t) = a_1a_2 \dots a_n$, $n \geq 1$, be a tree and its prefix notation, respectively. A *subtree jump table for prefix notation* $SJT\_pref(t)$ is defined as a mapping from a set $\{1..n\}$ into a set $\{2..n+1\}$. If $a_ia_{i+1} \dots a_{j-1}$ is the prefix notation of a subtree of tree t, then $SJT\_pref(t)[i] = j$, $1 \leq i < j \leq n+1$.

```
Name: ConstructSJT_pref
Input: Tree $t$ in prefix notation $pref(t)$ of size $n$
Input: Index of current node $rootIndex$ initialised to 1
Input/Output: Subtree jump table for prefix notation $SJT\_pref(t)$ initialised
              to empty array of size $n$
Result: Index $exitIndex$
1 begin
2     $index = rootIndex + 1$;
3     for $i = 1$ to $Arity(pref(t)[rootIndex])$ do
4         $index$ = ConstructSJT_pref($pref(t), index, SJT\_pref(t)$);
5     end
6     $SJT\_pref(t)[rootIndex] = index$;
7     return $index$;
8 end
```

Algorithm 17: Construction of subtree jump table for prefix notation.

Lemma 4.5.2. Given tree t in prefix notation $pref(t)$ and initial value of $rootIndex$ equal to 1, Algorithm 17 (ConstructSJT_pref) constructs subtree jump table for prefix notation $SJT\_pref(t)$.

Table 4.1: Subtree jump table for prefix notation for tree t_{2r} from Example 2.2.4.

node id	1	2	3	4	5	6	7	8	9	10	11	12	13
$SJT\_pref(t_{2r})$	14	11	8	5	6	7	8	9	10	11	12	13	14

Table 4.2: Array $Rev^{13}_{\{(1,11),(2,8),(3,5)\}}$ from Example 4.5.5.

node id	1	2	3	4	5	6	7	8	9	10	11	12	13
$Rev^{13}_{\{(1,11),(2,8),(3,5)\}}$	-1	-1	-1	-1	3	-1	-1	2	-1	-1	1	-1	-1

Informally, the subtree jump table contains an entry for each subtree r of tree t. The entry for subtree r is at the position of its root in the $pref(t)$ notation of the tree t. The entry stores an index one after the last symbol of the subtree r in $pref(t)$ notation of the tree t as a value. This structure has the same size as the prefix notation of the tree t.

Example 4.5.3. Consider tree t_{2r} over $\mathcal{A}$ from Example 2.2.4, $pref(t_{2r}) = a4\ a4\ a4\ a0\ b0\ a0\ a0\ a0\ b0\ a0\ a0\ a0\ b0$. With the indexing structure for locating substring of $pref(t_{2r})$ being a compact suffix automaton, the string index $FA_{csuf}(pref(t_{2r}))$ [21] is illustrated in Figure 2.9. Subtree jump table for prefix notation $SJT\_pref(t_{2r})$, constructed by Algorithm 17 (ConstructSJT_pref), is in Table 4.1. △

Furthermore, the array $Rev^n_{\mathcal{S}}$ serves as a working data structure for the main matching algorithm (Algorithm 20 (MatchPattern)) during the searching phase and its initial value, denoted $Rev^n_{\{\}}$, is to be set once and constructed during the preprocessing phase.

Definition 4.5.4. Let $\mathcal{S} = \{(first_1, last_1), \ldots (first_k, last_k)\}$ 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 $Rev^n_{\mathcal{S}}$ is an array of integers such that $Rev^n_{\mathcal{S}}[last_h] = first_h$ for all $1 \leq h \leq k$. For all other values $1 \leq v \leq n$, $Rev^n_{\mathcal{S}}[v] = -1$.

Example 4.5.5. Array $Rev^{13}_{\{(1,11),(2,8),(3,5)\}}$, which represents occurrences of a prefix $a4S$ of tree pattern p_{5r} from Example 2.2.6 in tree t_{2r} from Example 2.2.4, is illustrated in Table 4.2. △

4.5.2 Computing positions of all occurrences of a tree pattern

The section describes the searching phase using the index. Algorithm 20 (MatchPattern) computes the list of all occurrences of tree pattern p in the tree t. This main algorithm uses three algorithms, which are presented before Algorithm 20 (MatchPattern):

- Algorithm 18 (VerifyArityChecksum) [52] computes the arity checksum of tree pattern in prefix notation $pref(p)$ so that it would be verified that $pref(p)$ is a valid prefix notation of a tree pattern.

- A string index query algorithm corresponding to the selected string index. See Chapter 2 for examples.
- Algorithm 19 (MergeOccurrences) is an algorithm that merges two sets of occurrences in linear time. 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 [5]. Sorted list of occurrences of substrings cannot be obtained from the string index query algorithm in a linear time in general. Therefore, we have avoided such sorting completely, reaching the linear time with a special merge operation, which uses the working data structure Rev_S^n introduced in the previous section.

Definition 4.5.6. 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, $1 \le i \le k$, contains any symbol S. The substring p_i is called a *subpattern* of p at index i.

Example 4.5.7. Consider $pref(p_{5r}) = a4Sa0SS$, the prefix notation of tree pattern p_{5r} from Example 2.2.6. Tree pattern p_{5r} has four subpatterns, $pref(p_{5r}) = p_1 S p_2 S p_3 S p_4$, where $p_1 = a4$, $p_2 = a0$, $p_3 = \varepsilon$ and $p_4 = \varepsilon$. △

Definition 4.5.8. Let $pref(t) = a_1 a_2 \dots a_n$ be the prefix notation of a tree t. Let $pref(p) = p_1 S p_2 S \dots p_k$ be the prefix notation of a tree pattern p. An *occurrence* of subpattern p_i in $pref(t)$ is a pair $(first, last)$, where:

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

The set of all occurrences of subpattern 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 4.5.9. Consider subpattern $p_2 = a0$ of tree pattern p_{5r} from Example 2.2.6. Subpattern p_2 has seven occurrences in tree t_{2r} from Example 2.2.4: $occ^{t_{2r}}(p_2) = \{\ (4, 5)$, $(6, 7)$, $(7, 8)$, $(8, 9)$, $(10, 11)$, $(11, 12)$, $(12, 13)\ \}$. △

Definition 4.5.10. Let $pref(p) = p_1 S p_2 S \dots S p_k$ be the prefix notation of a tree pattern p. Then any string $p_1 S p_2 S \dots S p_{k'}$, $k' \le k$, or $p_1 S p_2 S \dots S p_{k''} S$, $k'' < k$, is called a *tree pattern prefix* of tree pattern p, abbreviated $TPP(p)$.

Example 4.5.11. Consider tree pattern p_{5r} and its prefix notation $pref(p_{5r}) = a4Sa0SS$, from Example 2.2.6. Then $\{a4, a4S, a4Sa0, a4Sa0S, a4Sa0SS\}$ is a set of tree pattern prefixes of tree pattern p_{5r}. △

Definition 4.5.12. Let p be a tree pattern and t be a tree. An *occurrence of tree pattern prefix* $TPP(p) = p_1 S p_2 S \dots S p_k$ in tree t is a pair $(first, last)$, where $(first, last_1)$ is an occurrence of subpattern p_1 in $pref(t)$, pair $(SJT\_pref(t)[last_1], last_2)$ is an occurrence of subpattern p_2 in $pref(t)$, …, and pair $(SJT\_pref(t)[last_{k-1}], last)$ is an occurrence of subpattern 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 4.5.13. Consider tree pattern prefix $TPP_1(p_{5r}) = a4S$ of tree pattern p_{5r} from Example 2.2.6. Consider tree t_{2r} from Example 2.2.4. Then $occ^{t_{2r}}(TPP_1(p_{5r})) = \{(1,11),(2,8),(3,5)\}$. $\triangle$

Lemma 4.5.14. 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}.

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 4.5.15. Consider tree pattern p_{5r} from Example 2.2.6. Tree pattern p_{5r} has two occurrences in tree t_{2r}: $occ^{t_{2r}}(p_{5r}) = \{(1,14),(2,11)\}$. $\triangle$

Lemma 4.5.16. 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 $last_A \neq last_B$.

```
Name: VerifyArityChecksum
Input: String over a ranked alphabet $str = a_1a_2 \dots a_n$, $n \geq 1$
Result: Decision whether $str = pref(t)$ for a tree $t$
1 begin
2     Set $ac(t) := 1$;
3     for $i := 1$ to $n$ do
4         $ac(str) := ac(str) + Arity(a_i) - 1$;
5         if $i < n$ and $ac(str) = 0$ then
6             return $false$;
7     end
8     return $ac(str) = 0$ ? $true$ : $false$;
9 end
```

Algorithm 18: Verification with the use of arity checksum [52].

Theorem 4.5.17. Let $TPP'(p) = p_1Sp_2S\dots Sp_{k-1}Sp_k$ 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 subpattern p_k. Given $prevOcc$ and $subOcc$ on input, Algorithm 19 (MergeOccurrences) computes occurrences $mergedOcc = occ^t(TPP'(p))$ of tree pattern prefix $TPP'(p)$.

```
Name: MergeOccurrences
Input: A set $prevOcc = occ^t(TPP(p))$, a set $subOcc = occ^t(p_k)$
Input: Temporary array $Rev_S^{|pref(t)|}$
Result: A set $mergedOcc = occ^t(TPP(p)p_k)$
begin
    $mergedOcc := \{\}$;
    foreach $(first, last)$ in $prevOcc$ do
        $Rev_{S \cup (first,last)}^{|pref(t)|}[last] := first$;
    end
    foreach $(first', last')$ in $subOcc$ do
        if $Rev_{prevOcc}^{|pref(t)|}[first'] \neq -1$ then
            $mergedOcc := mergedOcc \cup \{(Rev_{prevOcc}^{|pref(t)|}[first'], last')\}$;
        end
    end
    foreach $(first, last)$ in $prevOcc$ do
        $Rev_{S \setminus (first,last)}^{|pref(t)|}[last] := -1$;
    end
    return $mergedOcc$;
end
```

Algorithm 19: Merging Occurrences.

Example 4.5.18. Consider the prefix notation $pref(p_{5r}) = a4Sa0SS$ of tree pattern p_{5r}, illustrated in Figure 2.4. Tree pattern p_{5r} can be rewritten as $pref(p_{5r}) = p_1Sp_2Sp_3Sp_4$, where $p_1 = a4$, $p_2 = a0$ and $p_3 = p_4 = \varepsilon$.

We consider the run of Algorithm 20 (MatchPattern) using tree pattern p_{5r}, with the underlying string index used in the example chosen to be compact suffix automaton $FA_{csuf}(pref(t_{2r}))$, Algorithm 1 (FindOccurrences) as the string index query algorithm, and subtree jump table for prefix notation $SJT\_pref(t_{2r})$:

Algorithm 18 (VerifyArityChecksum) returns true for tree pattern p_{5r} because p_{5r} is a valid tree pattern (if you replaced S symbols with $a0$ symbols in the prefix notation of the pattern, you would get a prefix notation of a tree).

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

At $i = 2$, after Algorithm 1 (FindOccurrences) is executed, $occ = \{((4, 5), (6, 7), (7, 8), (8, 9), (10, 11), (11, 12), (12, 13)\}$. Using Algorithm 19 (MergeOccurences), $prevOcc$ is rewritten to $\{(1,\ 12),\ (2,\ 9)\}$. Using $SJT\_pref(t_{2r})$, $prevOcc$ is then rewritten to $prevOcc = \{(1,\ 13),\ (2,\ 10)\}$.

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

At $i = 4$, $prevOcc$ is not modified because subpattern p_4 is the empty string and the

```
Name: MatchPattern
Input: Tree pattern $pref(p) = p_1 S p_2 S \dots p_k$
Input: String index $SI(pref(t))$
Input: String index query algorithm $IndexQuery$ accepting string index
       $SI(pref(t))$ and string pattern $f$
Input: Subtree jump table for prefix notation $SJT\_pref(t)$
Input: Array $Rev_S^{|pref(t)|}$
Result: List of occurrences of tree pattern $p$
1  begin
2      if not VerifyArityChecksum(p) then
3          return ERROR – invalid pattern;
4      end
5      $prevOcc := \{\}$;
6      for $i := 1$ to $k$ do
7          if $p_i \neq \varepsilon$ then
8              $occ :=$ IndexQuery($SI(pref(t))$, $p_i$);
9              if $i = 1$ then $prevOcc := occ$;
10             else $prevOcc :=$ MergeOccurrences($prevOcc$, $occ$, $Rev_S^{|pref(t)|}$);
11         end
12         if $i \neq k$ then
13             foreach occurrence $(first, last)$ in $prevOcc$ do
14                 $(first, last) := (first, SJT\_pref(t)[last])$;
15             end
16         end
17     end
18     return $prevOcc$;
19 end
```

Algorithm 20: Searching for occurrences of a tree pattern.

algorithm returns set of occurrences $\{(1, 14), (2, 11)\}$.

Algorithm 20 (MatchPattern) has found two occurrences of tree pattern p_{5r}: the first one starting at position 1 (ending at position 14) and the second one at position 2 (ending at position 11) in $pref(t_{2r})$. $\triangle$

Theorem 4.5.19. Algorithm 20 (MatchPattern) finds all occurrences $occ^t(p)$ of tree pattern $p = p_1 S p_2 S \dots S p_k$ in tree t.

4.5.3 Time and space complexities

Lemma 4.5.20. Algorithm 17 (ConstructSJT_pref) runs in $\mathcal{O}(n)$ time, where n is the number of nodes of the subject tree t. Size of subtree jump table for prefix notation is n.

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 4.5.21. Construction of the index takes time $\mathcal{O}(n)$ time and produces the index of $\mathcal{O}(n)$ size.

Proof. The creation of the index structure depend on the creation time and size of the string index used within. Considering the $\mathcal{O}(n)$ time and space needed to construct the selected string index (like compact suffix automaton [21]) this requirement is met. Algorithm 17 (ConstructSJT_pref) that creates the subtree jump table for prefix notation is proved to be linear in time and space in Lemma 4.5.20. The array $Rev_{\mathcal{S}}$ is created in time $\mathcal{O}(n)$. □

Lemma 4.5.22. Algorithm 19 (MergeOccurrences) runs in $\mathcal{O}(|prevOcc| + |occ|)$ time, where $|prevOcc| + |occ|$ is the number of occurrences in both input sets.

Proof. The algorithm uses array $Rev_{\mathcal{S}}$ of size n prepared during the indexing phase. This array is used for the fast lookup. 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 4.5.23. Let $pref(p) = p_1Sp_2S\dots Sp_k$ of length m be the prefix notation of a tree pattern p. Algorithm 20 (MatchPattern) runs in $\mathcal{O}(m + \sum_{i=1}^{k} |occ'(p_i)|)$ time, where $occ'(p_i) = occ(p_i)$ if $p_i \neq \varepsilon$; otherwise, $occ'(p_i) = occ'(p_{i-1})$.

Proof. Verification of the arity checksum for the pattern takes $\mathcal{O}(m)$ time. Finding the occurrences of subpattern $p_i \neq \varepsilon$ takes time $\mathcal{O}(|p_i| + |occ(p_i)|)$. Summing over all subpatterns 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 19 (MergeOccurrences) with input size $\mathcal{O}(|occ(p_i)|)$, $p_i \neq \varepsilon$. Algorithm 19 (MergeOccurrences) 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 19 (MergeOccurrences) will be less than or equal to $\mathcal{O}(\sum_{i=1, p_i \neq \varepsilon}^{k} (2 * |occ(p_i)|)) = \mathcal{O}(\sum_{i=1}^{k} |occ'(p_i)|)$. □

4.6 Linear space index of a tree for nonlinear patterns

This section contains an original extension of the linear space index of a tree and the querying algorithm from the previous section to handle nonlinear tree templates as queries as well. The content is novel and not presented in any publication.

Table 4.3: Subtree repeats table for prefix notation for tree t_{2r} from Example 2.2.4.

node id	1	2	3	4	5	6	7	8	9	10	11	12	13
$SRT\_pref(t_{2r})$	5	4	3	1	2	1	1	1	2	1	1	1	2

4.6.1 Index modifications

The index additionally contains a table representing the subtree repeats to extend the ability of the aforementioned linear space index of a tree for tree patterns to answer nonlinear tree pattern queries. This table can be obtained by algorithm presented in [11] extended with an Algorithm 8 (PostfixToPrefixRepeats) presented in Section 3.4.

The summary of the index components consists of three parts, where the third is newly added:

- An arbitrary indexing structure constructed for the $pref(t)$, by which occurrences of all substrings of $pref(t)$ can be located. See Chapter 2 for examples.
- A *subtree jump table for prefix notation*, a linear-size structure needed for finding positions of ends of subtrees represented by special symbols S.
- A *subtree repeats table*, a linear-size structure for easy checking of subtree equivalence.

Example 4.6.1. Consider tree t_{2r} over $\mathcal{A}$ from Example 2.2.4, in prefix notation $pref(t_{2r}) = a4\ a4\ a4\ a0\ b0\ a0\ a0\ a0\ b0\ a0\ a0\ a0\ b0$. With the indexing structure for locating substring of $pref(t_{2r})$ being a compact suffix automaton, the string index $FA_{csuf}(pref(t_{2r}))$ [21] is illustrated in Figure 2.9. Subtree jump table for prefix notation $SJT\_pref(t_{2r})$, constructed by Algorithm 17 (ConstructSJT_pref), is unchanged in Table 4.1 in the previous section. Subtree repeats table for prefix notation, $SRT\_pref(t_{2r})$ constructed by Algorithm 8 (PostfixToPrefixRepeats), is in Table 4.3. $\triangle$

Furthermore, the array $Rev^n_{\mathcal{S}}$ is still used as a working data structure for the main matching algorithm. Additionally, some other arrays need to be used to maintain a fast search phase. Given the set of nonlinear variables $X, Y, Z, \ldots$, that can occur in the pattern, arrays denoted $Assign^n_{\mathcal{X},\mathcal{Y},\mathcal{Z},\ldots}$ are prepared to store information about subtree repeats. The initial value of these arrays is set again once in the preprocessing phase.

4.6.2 Querying algorithm modifications

The alphabet of the nonlinear tree pattern additionally contains some nonlinear variables $X, Y, Z, \ldots$. Therefore the definitions of subpattern need to be changed for nonlinear tree patterns.

Definition 4.6.2. Let $pref(p) = p_1 V_1 p_2 V_2 \ldots V_{k-1} p_k$ be the prefix notation of a nonlinear tree pattern p over an alphabet $A \cup \{S\} \cup \{X, Y, Z, \ldots\}$, where no substring p_i, $1 \le i \le k$, contains any symbol S, X, Y, Z, $\ldots$. The substring p_i is called a *subpattern* of p at index i. And where V_i, $1 \leq i \leq k-1$ is one of S, X, Y, Z, $\ldots$. The symbol V_i is called a *separator*.

Example 4.6.3. Consider $pref(p_{6r}) = a4Sa0XX$, the prefix notation of tree pattern p_{6r} from Example 2.2.6. Tree pattern p_{6r} has four subpatterns and three separators, $pref(p_{6r}) = p_1V_1p_2V_2p_3V_3p_4$, where $p_1 = a4$, $p_2 = a0$, $p_3 = \varepsilon$, $p_4 = \varepsilon$ and $V_1 = S$, $V_2 = X$, $V_3 = X$. $\triangle$

The support algorithms (Algorithm 18 (VerifyArityChecksum) and Algorithm 19 (MergeOccurrences)) can remain without changes, however the main matching algorithm from the previous section (Algorithm 20 (MatchPattern)) needs to be augmented to correctly handle the nonlinear tree patterns as Algorithm 21 (MatchNonlinearPattern).

The modification is present in a section of the algorithm where subtree skipping is performed. Nonlinear variables represent a complete subtrees, which are skipped the same way as in the previous case. However, the check for consistency of nonlinear variables setting in a particular occurrence needs to be present. Such a check is implemented using the array $Assign_{X,Y,Z,\ldots}^{|pref(t)|}$. The algorithm remembers a unique identifier of a subtree as represented in subtree repeats table $SRT\_pref(t)$ when a subtree is skipped by the first occurrence of a nonlinear variable. The algorithm later compares the subtree repeat identifier of a subtree skipped by next occurrences of the nonlinear variable with the remembered subtree repeat identifier.

The algorithm ensures the array $Assign_{X,Y,Z,\ldots}^{|pref(t)|}$ is cleared after each algorithm run similarly as the array $Rev_S^{|pref(t)|}$ is cleared.

Example 4.6.4. Consider the prefix notation $pref(p_{6r}) = a4Sa0XX$ of tree pattern p_{6r}, illustrated in Figure 2.4. Tree pattern p_{6r} can be rewritten as $pref(p_{6r}) = p_1V_1p_2V_2p_3V_3p_4$, where $p_1 = a4$, $p_2 = a0$, $p_3 = p_4 = \varepsilon$ and $V_1 = S$, $V_2 = V_3 = X$.

We consider the run of Algorithm 21 (MatchNonlinearPattern) using tree pattern p_{6r}, with compact suffix automaton $FA_{csuf}(pref(t_{2r}))$ chosen to be the underlying string index, Algorithm 1 (FindOccurrences) as the string index query algorithm, subtree jump table for prefix notation $SJT\_pref(t_{2r})$, and subtree repeats table $SRT\_pref(t)$ computed by algorithm presented in [11] with its result post-processed by Algorithm 8 (PostfixToPrefixRepeats):

Algorithm 18 (VerifyArityChecksum) returns true for tree pattern p_{6r} because p_{6r} is a valid tree pattern (if you replaced S and X symbols with $a0$ symbols in the prefix notation of the pattern, you would get a prefix notation of a tree).

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

At $i = 2$, after Algorithm 1 (FindOccurrences) is executed, $occ = \{((4,5), (6,7), (7,8), (8,9), (10,11), (11, 12), (12, 13)\}$. Using Algorithm 19 (MergeOccurences), $prevOcc$ is rewritten to $\{(1, 12), (2, 9)\}$. To represent the current setting of nonlinear variable X, the array $Array_X^{13}$ is modified as follows: $Array_X^{13}[1]$ is set to $SRT\_pref(t_{2r})[12]$ and

```
Name: MatchNonlinearPattern
Input: Tree pattern $pref(p) = p_1 V_1 p_2 V_2 \dots V_{k-1} p_k$
Input: String index $SI(pref(t))$
Input: String index query algorithm $IndexQuery$ accepting string index
       $SI(pref(t))$ and string pattern $f$
Input: Subtree jump table for prefix notation $SJT\_pref(t)$
Input: Subtree repeats table $SRT\_pref(t)$
Input: Array $Rev_{\mathcal{S}}^{|pref(t)|}$
Input: Array $Assign_{\mathcal{X},\mathcal{Y},\mathcal{Z},\dots}^{|pref(t)|}$
Result: List of occurrences of tree pattern $p$
1  begin
2      if not VerifyArityChecksum(p) then
3          return ERROR – invalid pattern;
4      end
5      $prevOcc := \{\}$;
6      for $i := 1$ to $k$ do
7          if $p_i \neq \varepsilon$ then
8              $occ :=$ IndexQuery($SI(pref(t))$, $p_i$);
9              if $i = 1$ then $prevOcc := occ$;
10             else $prevOcc :=$ MergeOccurrences($prevOcc$,$occ$,$Rev_{\mathcal{S}}^{|pref(t)|}$);
11         end
12         if $i \neq k$ then
13             foreach occurrence $(first, last)$ in $prevOcc$ do
14                 if $V_i \neq S$ and $Assign_{V_i}^{|pref(t)|}[first] \notin \{SRT\_pref(T)[last], -1\}$ then
15                     $(first, last) := (-1, last)$;
16                     $Assign_{X,Y,Z,\dots}^{|pref(t)|}[first] := -1$;
17                 else if $V_i \neq S$ and $Assign_{V_i}[first] = -1$ then
18                     $Assign_{V_i}^{|pref(t)|}[first] := SRT\_pref(T)[last]$;
19                 end
20                 $(first, last) := (first, SJT\_pref(t)[last])$;
21             end
22             $prevOcc := \{occ : occ \in prevOcc$ and $occ.first \neq -1\}$;
23         end
24     end
25     foreach occurrence $(first, last)$ in $prevOcc$ do
26         $Assign_{X,Y,Z,\dots}^{|pref(t)|}[first] := -1$;
27     end
28     return $prevOcc$;
29 end
```

Algorithm 21: Searching for occurrences of a nonlinear tree pattern.

$Array^{13}_X[2]$ is set to $SRT\_pref(t_{2r})[9]$. Using $SJT\_pref(t_{2r})$, $prevOcc$ is then rewritten to $prevOcc = \{(1, 13), (2, 10)\}$.

At $i = 3$, algorithm checks the equality of $Array^{13}_X[1]$ with $SRT\_pref(t_{2r})[13]$ and $Array^{13}_X[2]$ with $SRT\_pref(t_{2r})[10]$ concluding both differ. This is remembered by changing $prevOcc$ to $prevOcc = \{(-1, 13), (-1, 10)\}$. Again $SJT\_pref(t_{2r})$ is used to change $prevOcc$ to $prevOcc = \{(-1, 14), (-1, 11)\}$. And next the occurrences starting at position -1 are filtered out leaving $prevOcc = \{\}$.

At $i = 4$, $prevOcc$ is not modified because subpattern p_4 is the empty string and the algorithm returns empty set as a result.

Algorithm 21 (MatchNonlinearPattern) has correctly found no occurrence of tree pattern p_{6r} in t_{2r}. △

Theorem 4.6.5. Algorithm 21 (MatchNonlinearPattern) finds all occurrences $occ^t(p)$ of nonlinear tree pattern $p = p_1Sp_2S \dots Sp_k$ in tree t.

4.6.3 Time and space complexities

Theorem 4.6.6. Construction of the augmented index takes time $\mathcal{O}(n)$ time and produces index of $\mathcal{O}(n)$ size.

Proof. The augmented index is based on the index from the previous chapter of $\mathcal{O}(n)$ size which takes $\mathcal{O}(n)$ time to construct. It additionally contains a structure representing information about subtree repeats. The structure is an array of size $\mathcal{O}(n)$. Its construction time depends on the running time of the algorithm presented in [11] and possibly the transformation time of the subtree repeats table for postfix notation to a subtree repeats table for a different linear notation of a tree. Both the algorithm from [11] and the subtree repeats table transformation algorithm take $\mathcal{O}(n)$ time (see Theorem 3.4.4 or Theorem 3.4.6). Last the augmented index contains an array $Assign^n_{X,Y,Z,\dots}$ of size $\mathcal{O}(n \times |\{X, Y, Z, \dots\}|)$ $= \mathcal{O}(n)$, where $|\{X, Y, Z, \dots\}|$ is the number of nonlinear variables. □

Theorem 4.6.7. Let $pref(p) = p_1V_1p_2V_2 \dots V_{k-1}p_k$ of length m be the prefix notation of a tree pattern p. Algorithm 21 (MatchNonlinearPattern) runs in $\mathcal{O}(m + \sum_{i=1}^{k} |occ'(p_i)|)$ time, where $occ'(p_i) = occ(p_i)$ if $p_i \neq \varepsilon$; otherwise, $occ'(p_i) = occ'(p_{i-1})$.

Proof. The algorithm additionally performs the checking of nonlinear variable settings consistency. The check is performed along with prolonging the actual occurrence set with $SJT\_pref(t)$ table. The check for individual occurrence takes additional $\mathcal{O}(|\{X, Y, Z, \dots\}|) = \mathcal{O}(1)$ time and is performed at maximum $occ'(p_i)$ times for each subpattern p_i. Altogether the running time of the Algorithm 21 (MatchNonlinearPattern) is still $\mathcal{O}(\sum_{i=1}^{k} |occ'(p_i)|)$. □

4.7 Some empirical results

We have implemented our (nonlinear) tree indexing algorithms, ones based on the (nonlinear) full and linear index with three backends (position heap, suffix array, and compact suffix automaton) and pure automata based called (nonlinear) tree pattern pushdown automaton using C++ programming language. We compared our algorithms' performances against each other, by the running times of their construction and querying.

The comparison was done using a pattern set previously used for benchmarking Forest FIRE toolkit [12, 59]. This pattern set was obtained by taking the Mono project's X86 instruction set grammar and, for each grammar production, taking the tree in the production's right-hand side, and replacing any nonterminal occurrences by wildcard symbol occurrences or nonlinear variable symbol, obtaining a tree template and nonlinear tree template from each grammar production. The resulting two pattern sets, each, consists of 460 tree patterns of varying sizes (a mix of subtrees and tree templates and a mix of subtrees and nonlinear tree templates, respectively).

We indexed each subject tree from two sets of subject trees twice: a set of 150 trees of approximately 500 nodes each, and a set of 500 trees of approximately 150 nodes each. First, to answer queries including nonlinear tree templates and second to answer queries including tree templates. Resulting two sets of indexes were queried with all patterns from respective sets of patterns sequentially. Benchmarking was conducted on a 2 GHz Intel Core i7 with 16 GB of RAM running OpenSUSE GNU/Linux version Leap 15.0 using GCC version 7.3.1.

(Nonlinear) full and linear indexes are linear in size with respect to the indexed linearised versions of the subject trees, however, the automata-based index is exponential. Indexes are usually constructed to answer queries as fast as possible with less concern about their size. Because of that, the actual memory representation size was not measured.

Figures 4.10, 4.11, 4.12, and 4.13 show the query times to indexes for tree patterns (for two different subject tree sets) and query times to indexes for nonlinear tree patterns as boxplots (again for two different subject sets).

The measurements (note the logarithmic scale) clearly show that on average, the full and linear index with suffix array backend is the best among the backends of the full and linear indexes, and the (nonlinear) tree pattern automaton index despite its size (in its deterministic version) is the overall fastest answering index from the measured algorithms.

Figures 4.14, 4.15, 4.16, and 4.17 show the construction times of indexes for tree patterns (for two different subject tree sets) and construction times of indexes for nonlinear tree patterns (again for two different subject sets).

The measurements (again in the logarithmic scale) clearly show that on average, the full and linear index with suffix array backend is the fastest constructed among the backends of all indexes. The automata-based index is the slowest constructed, which corresponds to its theoretical size.

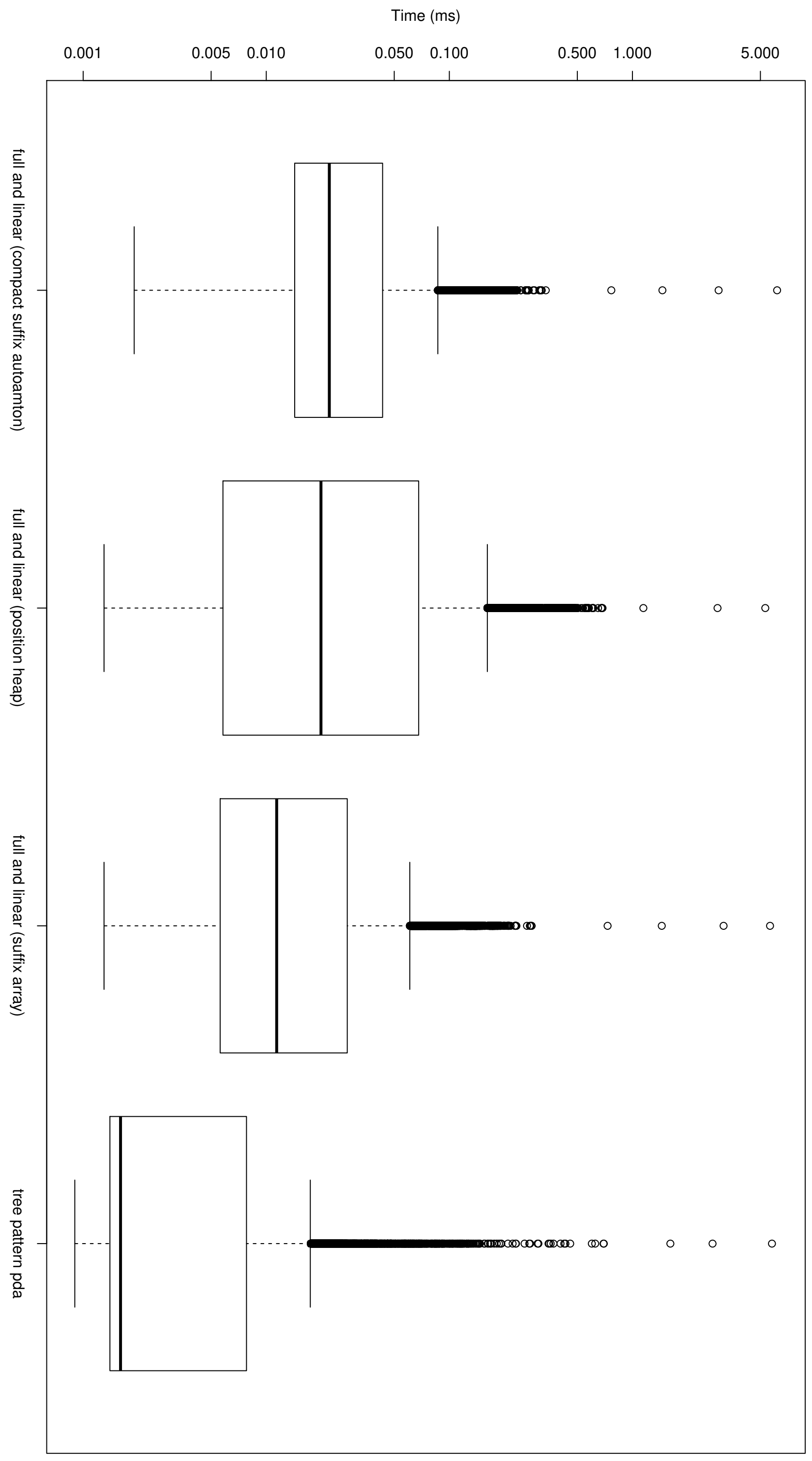


Figure 4.10: Distributions of querying times to tree pattern indexes on 150 trees of ca. 500 nodes each.

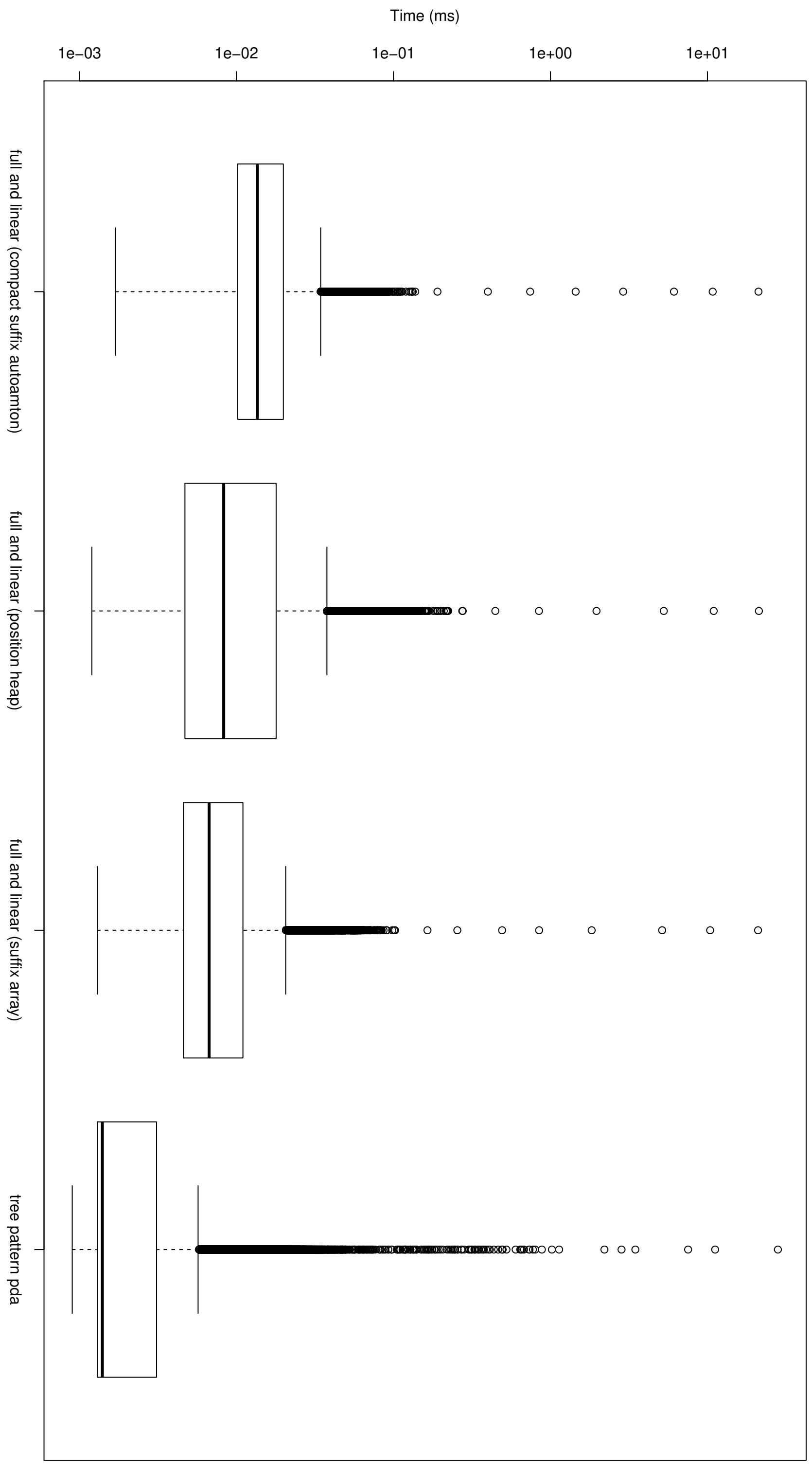


Figure 4.11: Distributions of querying times to tree pattern indexes on 500 trees of ca. 150 nodes each.

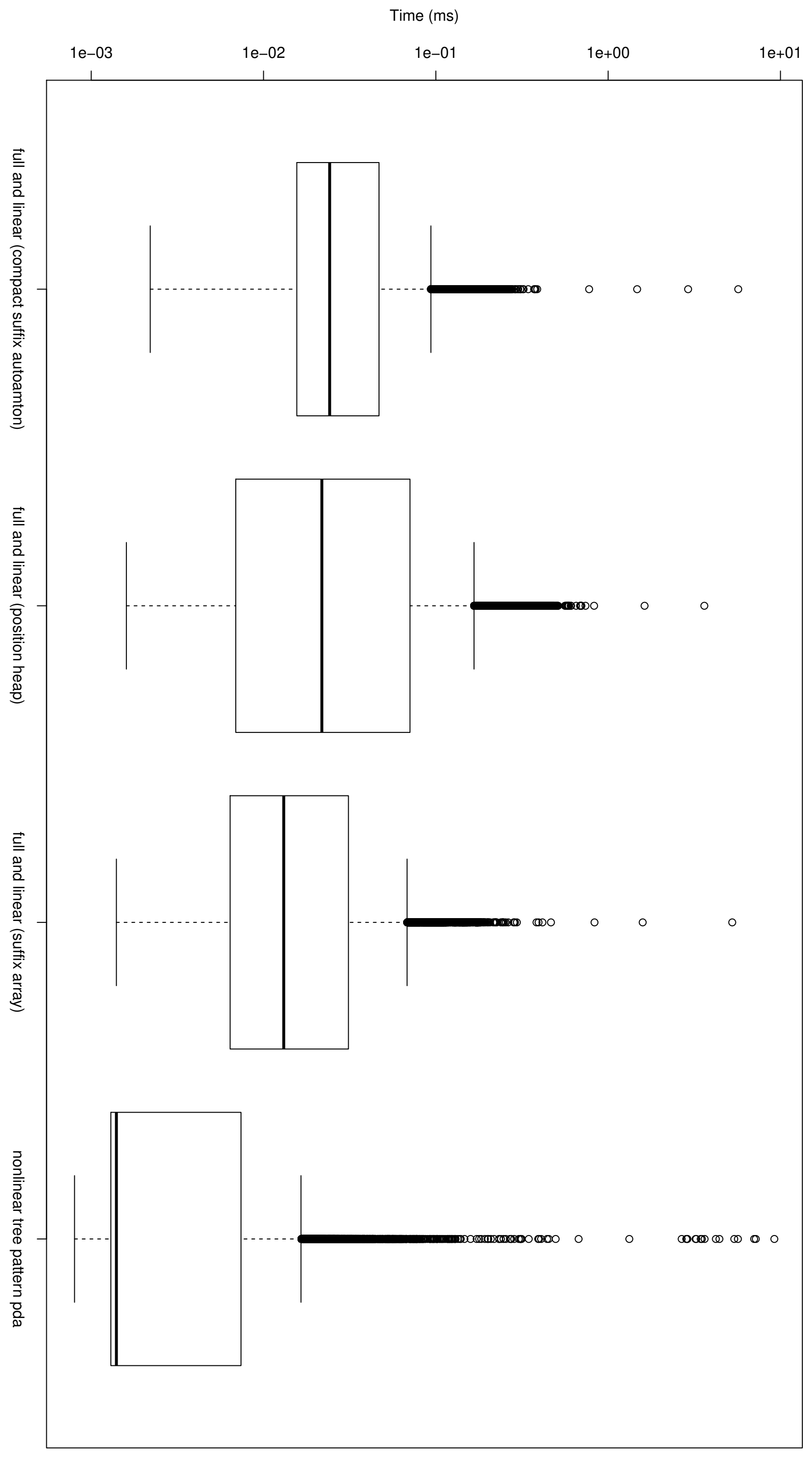


Figure 4.12: Distributions of querying times to nonlinear tree pattern indexes on 150 trees of ca. 500 nodes each.

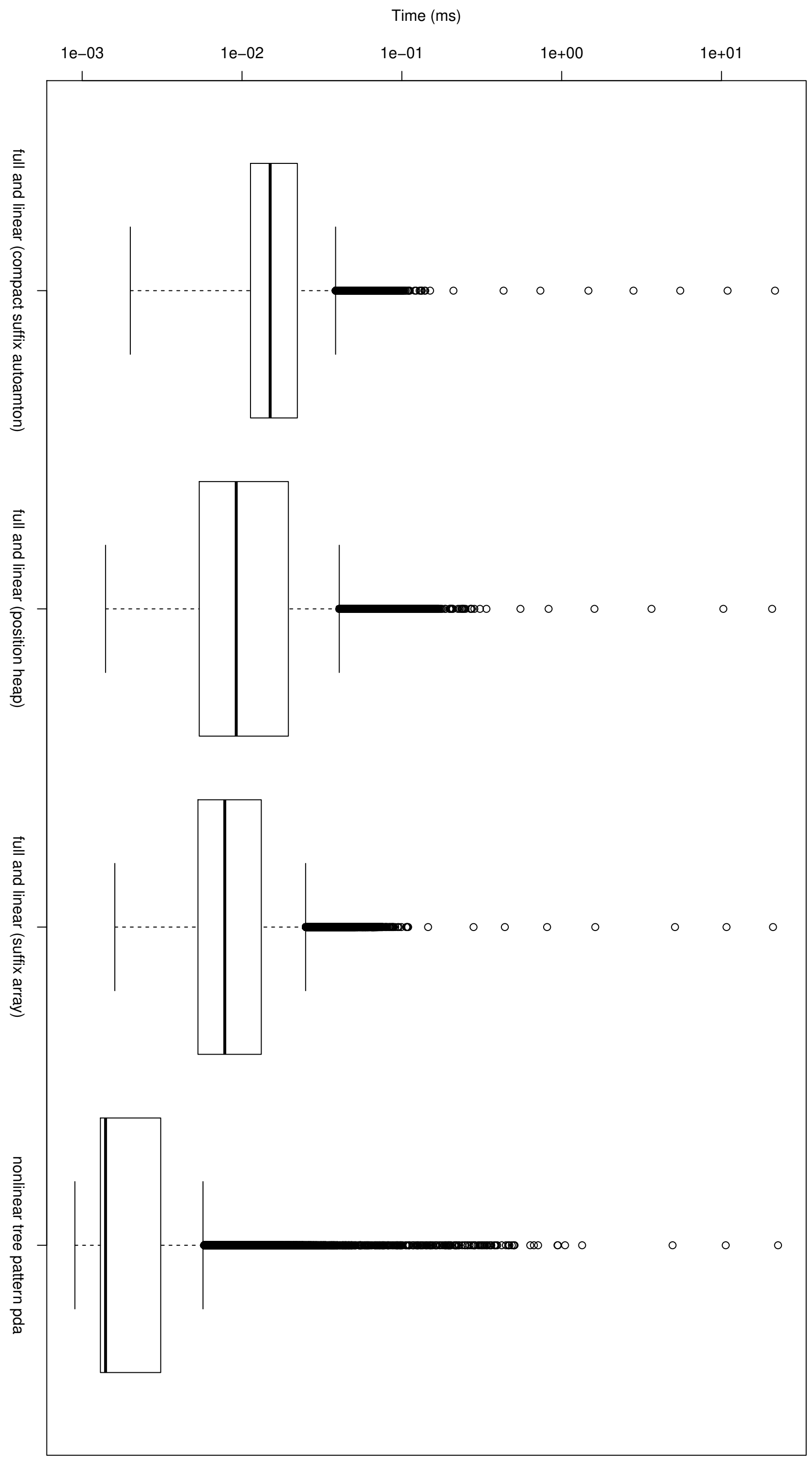


Figure 4.13: Distributions of querying times to nonlinear tree pattern indexes on 500 trees of ca. 150 nodes each.

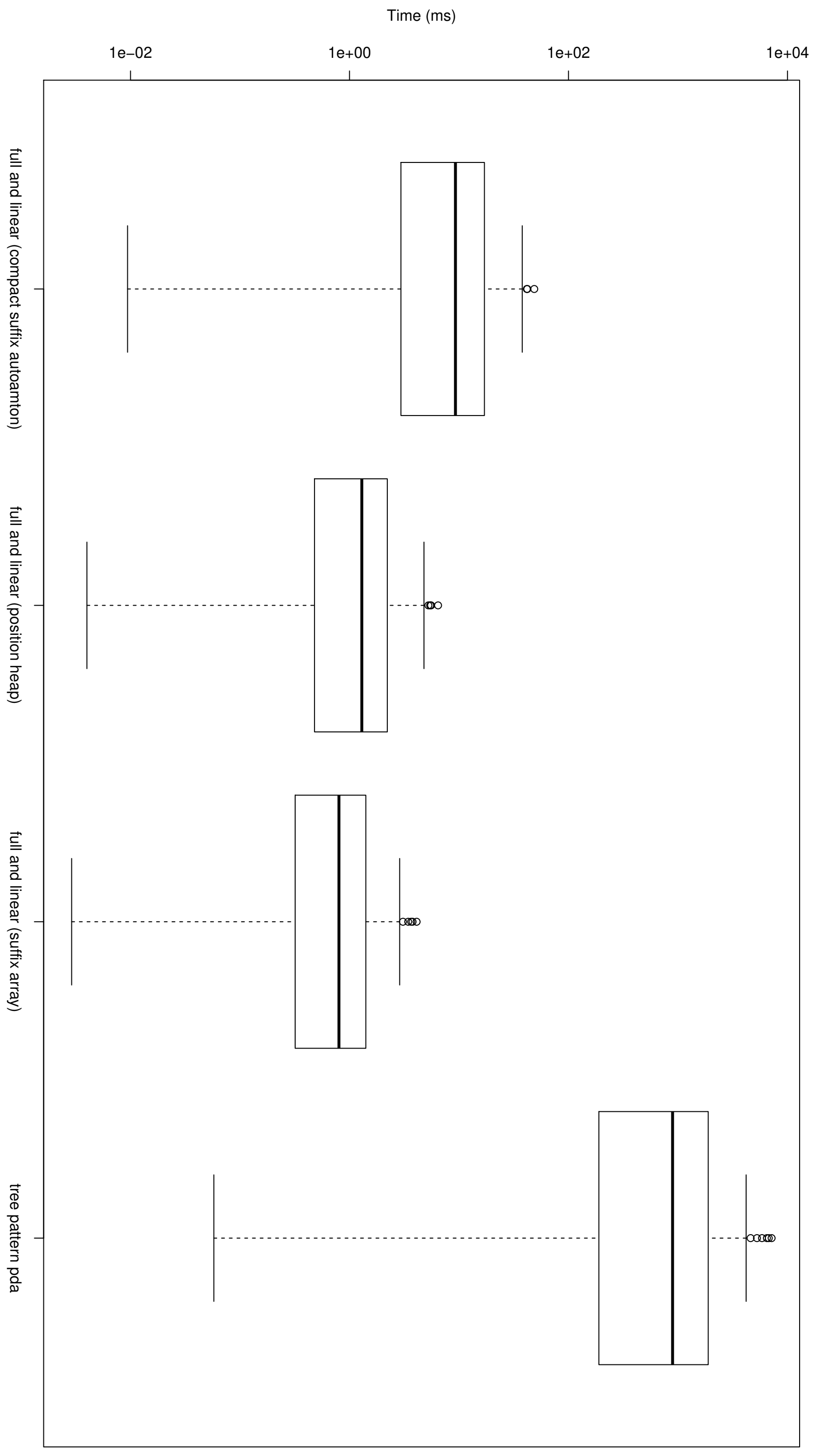


Figure 4.14: Distributions of index creation times of the tree pattern indexes on 150 trees of ca. 500 nodes each.

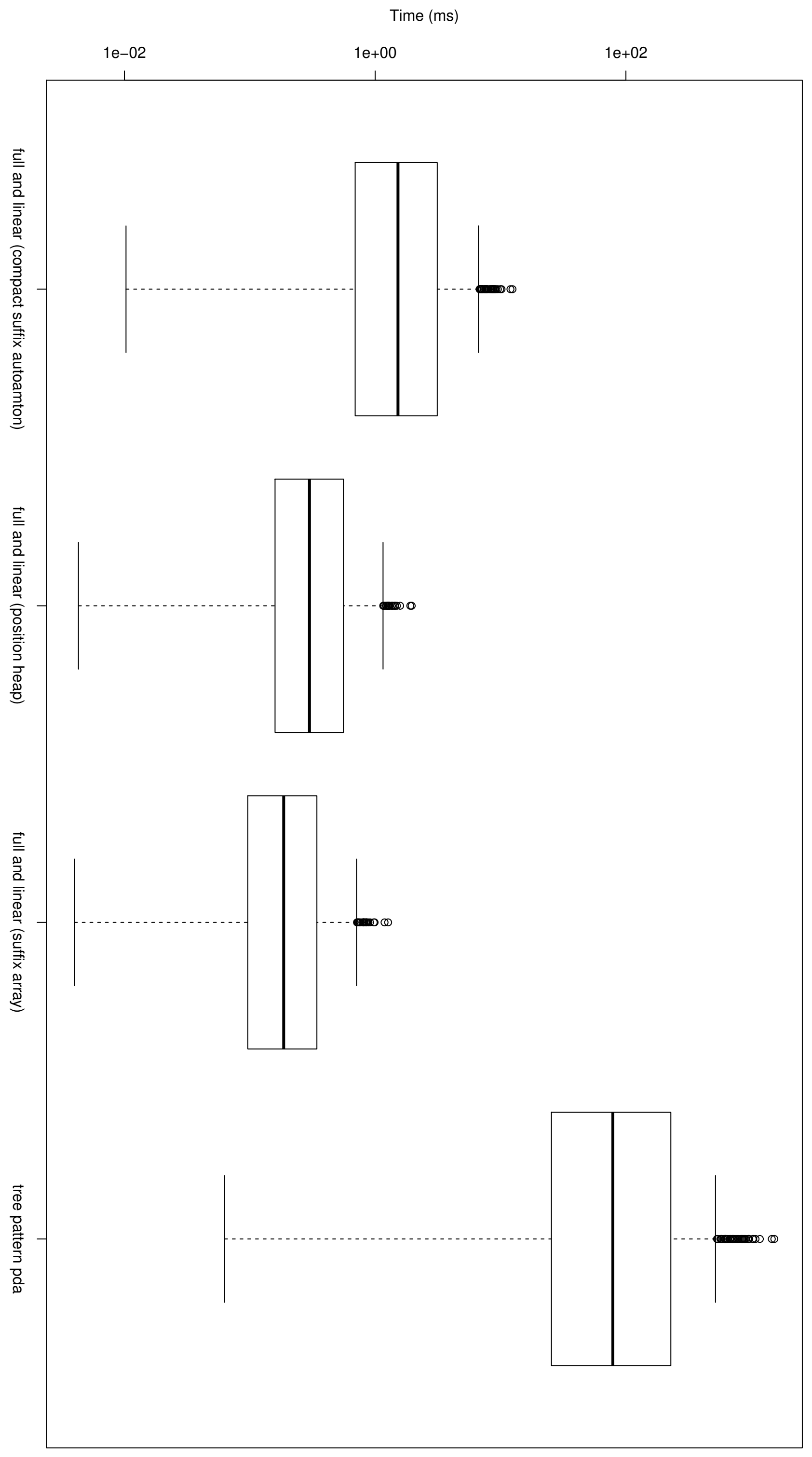


Figure 4.15: Distributions of index creation times of the tree pattern indexes on 500 trees of ca. 150 nodes each.

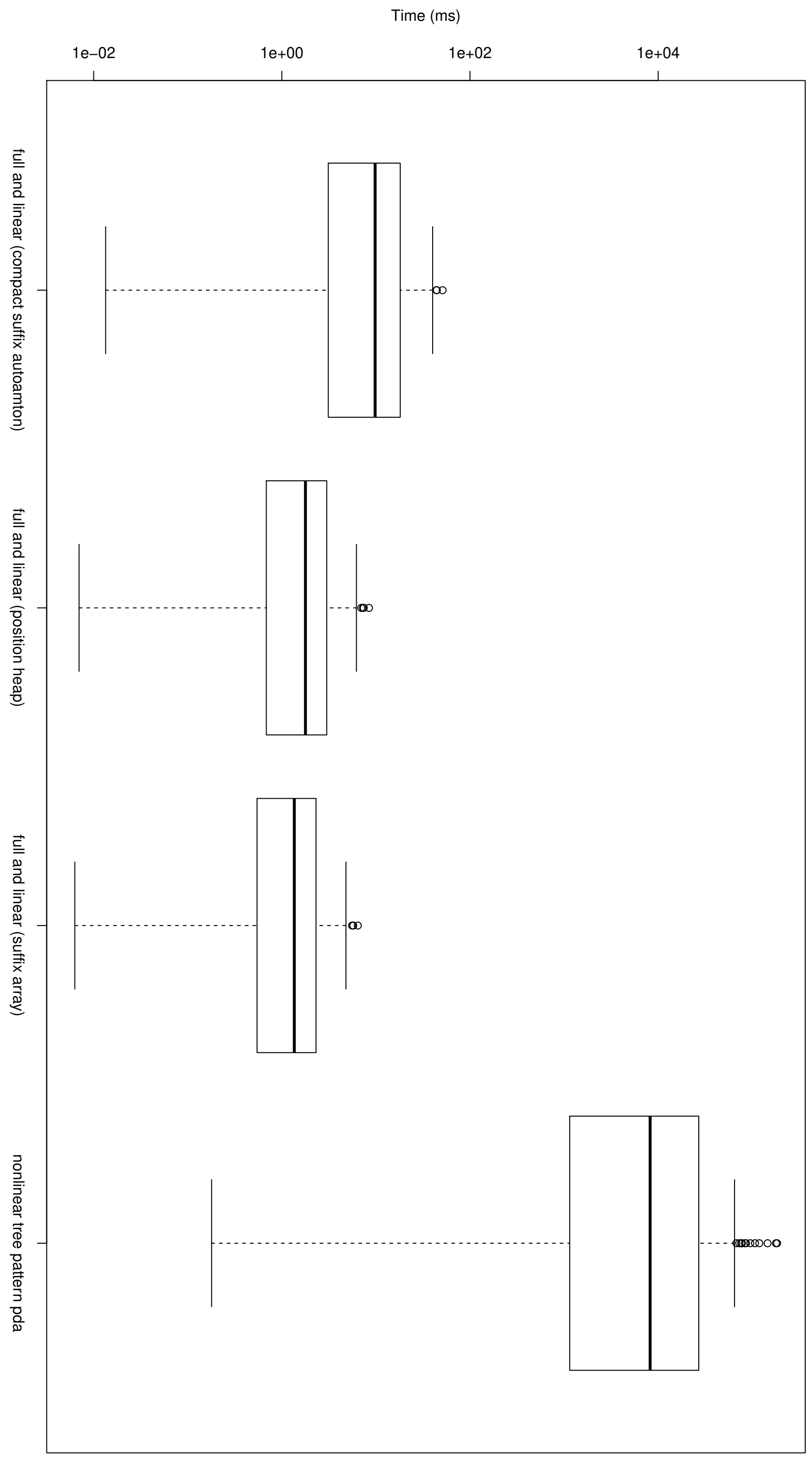


Figure 4.16: Distributions of index creation times of the nonlinear tree pattern indexes on 150 trees of ca. 500 nodes each.

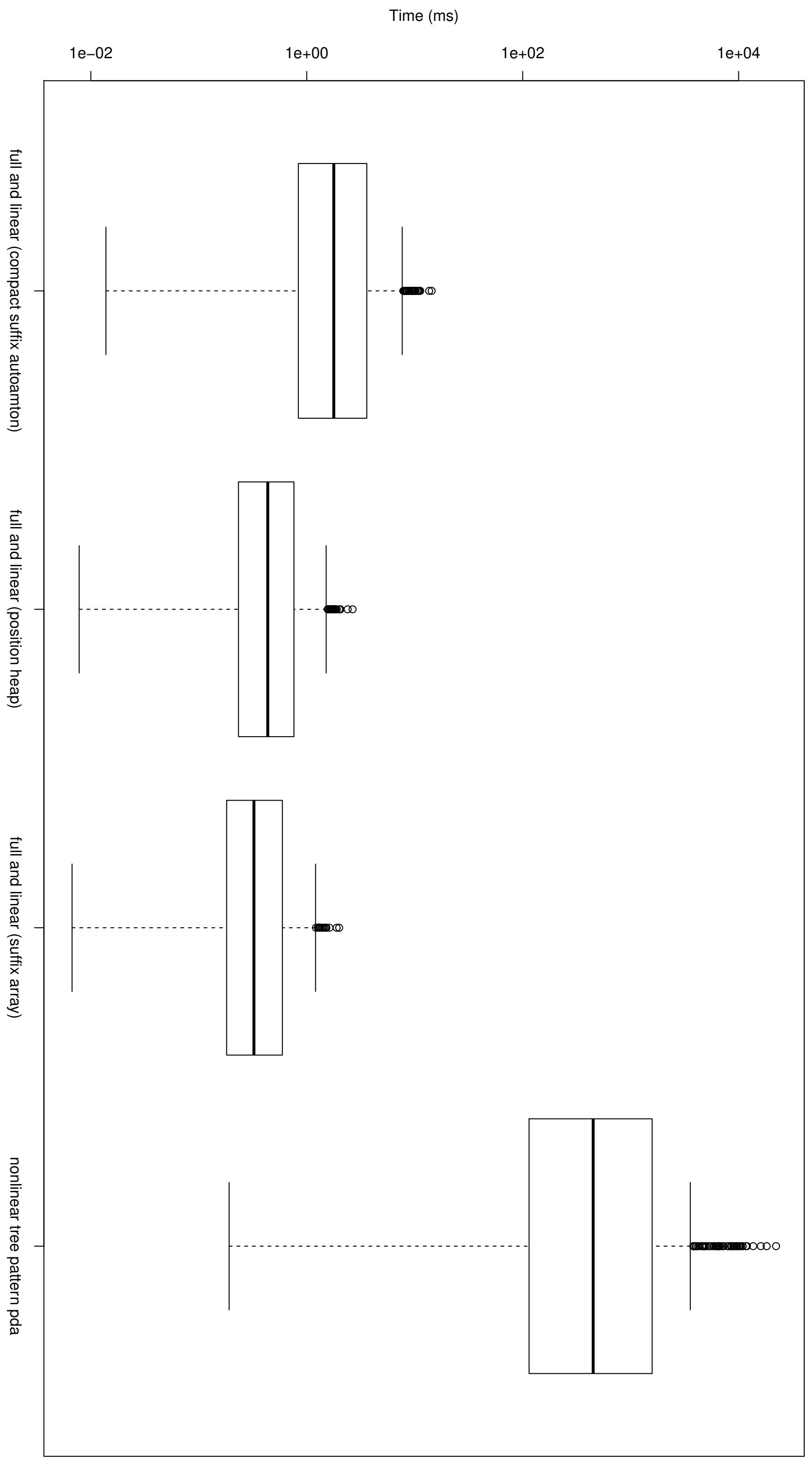


Figure 4.17: Distributions of index creation times of nonlinear tree pattern indexes on 500 trees of ca. 150 nodes each.

4.8 Conclusion of the tree indexing

We have presented the nonlinear tree pattern pushdown automaton, a new kind of pushdown automaton, which represents a complete index of a given tree for nonlinear tree patterns. Since the presented pushdown automaton is input–driven, it can be determinised.

Additionally, a new method of a full and linear index of a tree for tree patterns and nonlinear tree patterns has been presented. The presented algorithms can also be modified for unranked trees, however, such extension is not presented as it would only consist of an exchange of the used linear tree representation for prefix bar linear notation of the tree [52] instead of the prefix notation, or any other notation suitable for unranked trees.

We have discussed the time and space complexities of presented indexes. Based on experiments, the timings of our implementations of all presented indexes (deterministic version of the (nonlinear) tree pattern pushdown automata, full and linear index for (nonlinear) tree patterns with suffix array, compact suffix automaton, and position heap as string index backends) were shown.

For the future work, we would like to implement the full and linear index with more string index backends and provide a complete description of the full and linear family of indexes.

CHAPTER **5**

Main Results in Tree Pattern Matching

Tree pattern matching is covered by a family of algorithms designed for variety of tree linear notations. The chapter summary and experimental results is presented to conclude the tree pattern matching topic of this thesis.

5.1 Backward subtree matching

This section contains some supplementary results on subtree matching related to the topic of tree pattern matching. This section is included to make the results of tree pattern matching complete.

Subtrees of tree t are substrings of the tree t in all linear notations defined in this thesis and many others. Therefore any string pattern matching algorithm can be used to locate occurrences of subtrees inside a subject tree without changes. However, not all substrings of tree t in these linear notations are subtrees of the tree t. The linear notations of trees are strings with additional properties that can still be used to improve existing string pattern matching algorithms.

Theorem 5.1.1. Given the prefix notation $pref(t)$ of a tree t, the borders of the $pref(t)$ are ε and $pref(t)$ only.

Proof. $pref(t)$ is the prefix notation of a subtree of t, if and only if arity checksum $ac(w) = 0$. Let w_1 be a prefix of w, where $w_1 \neq w$ and $w_1 \neq \varepsilon$. The $ac(w_1) \geq 1$. Let w_2 be a suffix of w, where $w_2 \neq w$ and $w_2 \neq \varepsilon$. Given the $ac(w) = 0$, the $ac(w_2) = ac(w) - ac(w_1) \leq 0$ for each w_2, where $w = w_1w_2$. Border is a prefix that is also a suffix. Clearly no prefix w_1, where $w_1 \neq \varepsilon$ and $w_1 \neq pref(t)$, with $ac(w_1) \geq 1$ can also be a suffix w_2 with $ac(w_2) \leq 0$. □

The same theorem also holds for bar notation of a tree $pref\_bar(t)$ and notations defined further in the thesis.

Even though trees are in general generated by context-free grammars (or tree grammars), finite automata can be used to locate the occurrences of a subtree inside a subject tree.

String pattern matching algorithms based on shifting of the pattern and skipping some comparisons of symbols, like the Boyer-Moore or Boyer-Moore-Horspool algorithms, can't check whether their input is, in fact, a tree in linear notation (if such a check is required). The check would need to be done explicitly with the arity checksum execution either before, after, or together with the pattern matching algorithm. The pattern matching algorithms themselves may achieve sublinear complexity with respect to the size of the subject string. Executing the arity checksum is, however, linear with respect to the size of the tree. Therefore the overall complexity would not be better than linear with respect to the size of the subject tree.

5.2 Backward linearised tree pattern matching algorithm

This section contains base results from an individual work presented as a conference paper [61].

The problem of tree pattern matching can be seen as matching of connected subgraphs in trees. Tree patterns in a linear notation are represented by substrings of trees in the linear notation but they can contain gaps given by wildcards S, which serve as placeholders for any subtree. The gaps are of variable length but always contain a complete subtree. A variant of the subtree jump table introduced in the previous chapter will be used to efficiently skip these gaps.

The basic idea of backward linearised tree pattern matching for tree patterns is the same as in the string case: shift the pattern in one direction and match symbols of the tree pattern and the subject tree in the opposite direction. Wildcard S occurrences must be handled specially. First we present the algorithm using a prefix ranked bar notation of the subject tree and the tree pattern.

Definition 5.2.1. The *prefix ranked bar notation* $pref\_ranked\_bar(t)$ of a tree t is defined as follows:

1. $pref\_ranked\_bar(S) = S \uparrow S$
2. $pref\_ranked\_bar(a) = a0 \uparrow 0$ if a is a leaf,
3. $pref\_ranked\_bar(t) = an\ pref\_ranked\_bar(b_1) \dots pref\_ranked\_bar(b_n) \uparrow n$, where a is the root of the tree t, $n = Arity(a)$ and $b_1, \dots\ b_n$ are direct descendants of a.

Prefix ranked bar notation is a combination of the prefix notation and the bar notation. The prefix ranked bar notation has a following useful property inherited from the bar notation: both the beginning and the end of occurrence of each subtree in the tree has its unique representation. The beginning is represented by a non-bar symbol and the end is represented by a bar symbol. The arity inherited from the prefix notation extends the alphabet which in turn extend the shifts. The size of the prefix ranked bar notation

Table 5.1: Prefix, postfix, prefix bar, postfix bar, and prefix ranked bar linear notations of tree t_{1r}.

prefix_bar	a	a	a	$\uparrow$	a	a	$\uparrow$	$\uparrow$	$\uparrow$	a	a	$\uparrow$	$\uparrow$	$\uparrow$
postfix_bar	$\uparrow$	$\uparrow$	$\uparrow$	a	$\uparrow$	$\uparrow$	a	a	a	$\uparrow$	$\uparrow$	a	a	a
prefix	$a2$	$a2$	$a0$		$a1$	$a0$				$a1$	$a0$			
postfix				$a0$			$a0$	$a1$	$a2$			$a0$	$a1$	$a2$
prefix_ranked_bar	$a2$	$a2$	$a0$	$\uparrow 0$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$

is exactly $2 * n$, where n is the size of the tree, which on the other hand makes the representation of the tree longer.

Example 5.2.2. Consider tree t_{1r} from Example 2.2.1 and its prefix, postfix, prefix bar, postfix bar, and prefix ranked bar notations. The linear notations of tree t_{1r} are all together illustrated in Table 5.1. Table 5.1 is an extension of Table 2.1 from Example 2.2.3. $\triangle$

Definition 5.2.3. Let $\uparrow n$, where $n \geq 0$ be bar symbols of arity n. The *bar set* $\mathcal{A}_\uparrow$ is the set of all bar symbol $\uparrow n$.

Note that every subtree in the prefix ranked bar notation ends with some symbol from $\uparrow n$, where $n \geq 0$ and starts with a symbol that is not in set $\uparrow n$.

5.2.1 Construction of bad character shift table

Definition 5.2.4. Let $pattern[1..m]$ be a *pref_ranked_bar* notation of tree pattern p over alphabet $\mathcal{A}$. The *bad character shift table* $BCS\_pref\_ranked\_bar(p)$ for the backward linearised tree pattern matching is defined for each $a \in \mathcal{A}$.

$$BCS\_pref\_ranked\_bar(p)[a] =$$
$$\min \left(\begin{array}{l} \{m\} \cup \{j : pattern[m-j] = a \text{ and } m > j > 0\} \cup \\ \{j + Arity(a) * 2 : pattern[m-j] = S \text{ and } m > j > 0 \text{ and } a \notin \mathcal{A}_\uparrow\} \cup \\ \{j - 1 : pattern[m-j] = S \text{ and } m > j > 0 \text{ and } a \in \mathcal{A}_\uparrow \text{ and } pattern \neq S \uparrow S\} \cup \\ \{1 : a \in \mathcal{A}_\uparrow \text{ and } pattern = S \uparrow S\} \end{array} \right)$$

Note that there is no value for wildcard S in the bad character shift table. Wildcard S cannot occur in the subject tree, therefore the value for it does not need to be in the bad character shift table.

Items of the bad character shift table are computed as the minimum value from four formulas, see Definition 5.2.4 where the formulas are separated by the union operation. The first formula makes sure that the shift is not longer than the size of pattern m. In this case, the size of a subtree corresponding to wildcard S is considered to be the smallest possible one, i.e. 2 (the size of subtree consisting of one nullary symbol $a0 \uparrow 0$). The second formula defines the minimal safe shift for symbols that occur in the pattern. The minimal safe shift for symbol a is the distance j of the closest occurrence of symbol a from the end of the pattern. Nullary symbol S is considered to correspond to the smallest possible subtree again.

The third and fourth formulas define the shift for cases when symbol a is expected to be inside subtree t_e that corresponds to wildcard S. The location of the last wildcard S from the end of the pattern is used to define the base shift length j and this shift can be prolonged by some number depending on the arity of symbol a, see the second part of the union in the definition. The smallest subtree t_e that contains symbol a is rooted by a and its direct descendants are nullary symbols $b0$. For each symbol $b0$ in subtree t_e there is also one symbol $\uparrow 0$. The base shift j is then prolonged by $2 * Arity(a)$. Any symbol from set $\mathcal{A}_\uparrow$ can occur as the last symbol of subtree t_e, i.e. it can be matched with $\uparrow S$. Therefore, the base shift of each bar is shortened by 1, see the fourth part of the union in the definition. The definition is extended to handle the $S \uparrow S$, which, despite not being allowed, is handled correctly by the fourth and fifth part of the union in the definition.

```
Name: ConstructBCS_pref_ranked_bar
Input: Tree pattern in the prefix ranked bar notation $pref\_ranked\_bar(pattern)$
       of size $m$ over alphabet $\mathcal{A}$ of the subject tree
Result: The bad character shift table $BCS\_pref\_ranked\_bar(pattern)$
begin
    $s := m$;
    for $i := 1$ to $m$ do
        if $pref\_ranked\_bar(pattern)[i] = S$ then $s = m - i$;
    end
    foreach $x \in \mathcal{A}$ do $BCS\_pref\_ranked\_bar(pattern)[x] = m$;
    foreach $x \in \mathcal{A}$ do
        if $x \notin \mathcal{A}_\uparrow$ then $shift := s + Arity(x) * 2$;
        else if $s >= 2$ then $shift := s - 1$;
        else $shift := s$;
        if $BCS\_pref\_ranked\_bar(pattern)[x] > shift$ then
          $BCS\_pref\_ranked\_bar(pattern)[x] := shift$;
    end
    for $i := 1$ to $m - 1$ do
        if $pref\_ranked\_bar(pattern)[i] \notin \{S, \uparrow S\}$ and
          $BCS\_pref\_ranked\_bar(pattern)[pref\_ranked\_bar(pattern)[i]] > (m - i)$
          then
          $BCS\_pref\_ranked\_bar(pattern)[pref\_ranked\_bar(pattern)[i]] := m - i$;
    end
    return $BCS\_pref\_ranked\_bar(pattern)$;
end
```

Algorithm 22: Construction of the bad character shift table.

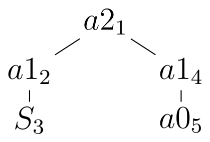


Figure 5.1: Tree pattern p_{8r} from Example 5.2.5.

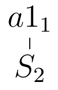


Figure 5.2: Tree pattern p_{9r} from Example 5.2.6.

Firstly, Algorithm 22 (ConstructBCS_pref_ranked_bar) for the construction of the bad character shift table finds the location of the last wildcard S. Then, the bad character shift table entries are initialised to the size of the pattern for all symbols of the alphabet. The length of the shift for all symbols of the alphabet is possibly shortened with the use of the information on the position of the last wildcard S in the pattern. The arity of symbols is used to make this part of the shift function longer according to Definition 5.2.4. Finally, the length of the shift is again possibly shortened by the actual positions of symbols in the pattern.

Example 5.2.5. Consider a tree pattern p_{8r}, depicted in Figure 5.1, in the prefix ranked bar notation $pref\_ranked\_bar(p_{8r}) = a2\ a1\ S\ {\uparrow}S\ {\uparrow}1\ a1\ a0\ {\uparrow}0\ {\uparrow}1\ {\uparrow}2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, {\uparrow}3, {\uparrow}2, {\uparrow}1, {\uparrow}0, {\uparrow}S\}$. Algorithm 22 (ConstructBCS_pref_ranked_bar) constructs the following items of the bad character shift table $BCS\_pref\_ranked\_bar(p_{8r})$ abbreviated as BCS. △

$$\begin{array}{ll}
BCS[a3] = \min(\{10\} \cup \emptyset \cup \{13\}) = 10 & BCS[a2] = \min(\{10\} \cup \{9\} \cup \{11\}) = 9 \\
BCS[a1] = \min(\{10\} \cup \{4, 8\} \cup \{9\}) = 4 & BCS[a0] = \min(\{10\} \cup \{3\} \cup \{7\}) = 3 \\
BCS[{\uparrow}3] = \min(\{10\} \cup \emptyset \cup \{6\}) = 6 & BCS[{\uparrow}2] = \min(\{10\} \cup \emptyset \cup \{6\}) = 6 \\
BCS[{\uparrow}1] = \min(\{10\} \cup \{1, 5\} \cup \{6\}) = 1 & BCS[{\uparrow}0] = \min(\{10\} \cup \{2\} \cup \{6\}) = 2
\end{array}$$

Example 5.2.6. Consider tree pattern p_{9r} in linear notation $pref\_ranked\_bar(p_{9r}) = a1\ S\ {\uparrow}S\ {\uparrow}1$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, {\uparrow}3, {\uparrow}2, {\uparrow}1, {\uparrow}0, {\uparrow}S\}$ depicted in Figure 5.2. Shifts for wildcard symbol S are described in Figure 5.3. The case of the shift for ${\uparrow}0$ is 1, because ${\uparrow}0$ is aligned to symbol ${\uparrow}S$. Other cases of shifts for symbols not in $\mathcal{A}_{\uparrow}$ are lengthened by their arity. △

5.2.2 Construction of subtree jump table for bar notation

The backward linearised tree pattern matching algorithm uses an additional structure subtree jump table for bar notation (SJT_bar) to efficiently skip subtrees corresponding to S. The subtree jump table for bar notation is similar to the subtree jump table for

TEXT	$\boxed{a1a1a1a3a0{\uparrow}0a0{\uparrow}0a0{\uparrow}0{\uparrow}3{\uparrow}1{\uparrow}1{\uparrow}1}$		$\boxed{a1a1a1a1a0{\uparrow}0{\uparrow}1{\uparrow}1{\uparrow}1{\uparrow}1}$
PATTERN	$\boxed{a1\,S\,{\uparrow}S{\uparrow}1}$		$\boxed{a1\,S\,{\uparrow}S{\uparrow}1}$
S-SHIFT	$\boxed{a1S \rightarrow \qquad \leftarrow S{\uparrow}1}$		$\boxed{a1S \rightarrow \leftarrow S{\uparrow}1}$
TEXT	$\boxed{a1a1a1a2a0{\uparrow}0a0{\uparrow}0{\uparrow}2{\uparrow}1{\uparrow}1{\uparrow}1}$	$\boxed{a1a1a1a0{\uparrow}0{\uparrow}1{\uparrow}1{\uparrow}1}$	$\boxed{a1a1a0{\uparrow}0{\uparrow}1{\uparrow}1}$
PATTERN	$\boxed{a1\,S\,{\uparrow}S{\uparrow}1}$	$\boxed{a1\,S\,{\uparrow}S{\uparrow}1}$	$\boxed{a1\,S\,{\uparrow}S{\uparrow}1}$
S-SHIFT	$\boxed{a1S \rightarrow \qquad \leftarrow S{\uparrow}1}$	$\boxed{a1\,S\,{\uparrow}S{\uparrow}1}$	$\boxed{a1S\,{\uparrow}S{\uparrow}1}$

Figure 5.3: Subtree shifts for tree pattern p_{9r} from Example 5.2.6.

prefix notation introduced in Definition 4.5.1. The subtree jump table for bar notation allows jumping over subtrees in both directions. There is a jump before the subtrees on positions corresponding to bar symbols and after the subtree on positions corresponding to non-bar symbols. Therefore the subtree jump table SJT_bar encodes jumps over subtrees for both prefix and postfix bar notations, or any other notation that share the structure of the notation, like prefix ranked bar notation.

Definition 5.2.7. Let t and $pref\_ranked\_bar(t)$ of length n be a tree and its prefix ranked bar notation, respectively. A *subtree jump table for bar notation* denoted as $SJT\_bar(t)$ is defined as a mapping from a set of integers $\{1..n\}$ into a set of integers $\{0..n+1\}$. If $pref\_ranked\_bar(t)\ [i..j]$ is the $pref\_ranked\_bar(s)$ notation of a subtree s of a tree t, then the $SJT\_bar(t)[i] = j + 1$ and $SJT\_bar(t)\ [j] = i - 1$, $1 \leq i < j \leq n$.

Informally, this variant of subtree jump table contains two entries for each subtree r of tree t. The first entry is at the position of the first symbol of the subtree r in $pref\_ranked\_bar(r)$ notation in the $pref\_ranked\_bar(t)$ notation of the tree t, and it stores an index one after the last symbol of the subtree r in $pref\_ranked\_bar(r)$ notation as a value. The second entry is at the position of the last symbol of the subtree r in $pref\_ranked\_bar(r)$ notation in the $pref\_ranked\_bar(t)$ notation of the tree t and it stores an index one before the first symbol of the subtree r in $pref\_ranked\_bar(r)$ notation as a value. This structure has the same size as the $pref\_ranked\_bar$ notation of the tree t and it can be construction by Algorithm 23 (ConstructSJT_bar).

Lemma 5.2.8. Given tree t in prefix ranked bar notation $pref\_ranked\_bar(t)$ and initial value of $rootIndex$ equal to 1, Algorithm 23 (ConstructSJT_bar) constructs subtree jump table for bar notation $SJT\_bar(t)$.

Example 5.2.9. Consider a tree t_{4r} in the prefix ranked bar notation $pref\_ranked\_bar$ (t_{4r}) = $a2$ $a2$ $a0$ $\uparrow 0$ $a0$ $\uparrow 0$ $\uparrow 2$ $a2$ $a0$ $\uparrow 0$ $a0$ $\uparrow 0$ $\uparrow 2$ $\uparrow 2$ over alphabet $\mathcal{A} = \{a3, a2, a1, a0, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0\}$. Table 5.2 shows the $SJT\_bar(t_{4r})$. △

```
Name: ConstructSJT_bar
Input: Tree $t$ in prefix notation $pref\_ranked\_bar(t)$ of length $n$
Input: Index of current node $rootIndex$ initialised to 1
Input/Output: Subtree jump table for bar notation $SJT\_bar(t)$ initialised to
              empty array of size $n$
Result: Index $exitIndex$
begin
    $index := rootIndex + 1$;
    for $i = 1$ to $Arity(pref\_ranked\_bar(t)[rootIndex])$ do
        $index := ConstructSJT\_bar(pref\_ranked\_bar(t), index, SJT\_bar(t))$;
    end
    $index := index + 1$;
    $SJT\_bar(t)[rootIndex] = index$;
    $SJT\_bar(t)[index - 1] = rootIndex - 1$;
    return $index$;
end
```

Algorithm 23: Construction of subtree jump table for bar notation.

Table 5.2: Subtree jump table for bar notation $SJT\_bar(t_{4r})$ of tree t_{4r}.

node id	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$pref(t_{4r})$	$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$
$SJT\_bar(t_{4r})$	15	8	5	2	7	4	1	14	11	8	13	10	7	0

5.2.3 The backward tree pattern matching algorithm

Our backward linearised tree pattern matching algorithm, presented as Algorithm 24 (BackwardLTPM), is an extension of the string backward pattern matching algorithm, presented as Algorithm 2 (BasicBackwardPatternMatchingAlgorithm).

The modification of the string backward matching algorithm is based on the principle that the algorithm also performs tests for wildcards S in the pattern. The modification is on lines 9 to 11 of Algorithm 24 (BackwardLTPM), where a part of the subject tree representing a subtree is skipped when a wildcard S, represented as $S \uparrow S$, is processed. Also, two indexes, one to the pattern and the other one to the text, are needed because subtrees (which need to be skipped) are often longer than two symbols.

Theorem 5.2.10. Given tree pattern p in the prefix ranked bar notation and bad character shift table $BCS\_pref\_ranked\_bar(p)$ constructed for the tree pattern p by Algorithm 22 (ConstructBCS_pref_ranked_bar), Algorithm 24 (BackwardLTPM) correctly computes the locations of all occurrences of tree pattern p in subject tree t.

Proof. The backward linearised tree pattern matching algorithm for prefix ranked bar notation is an extension of the backward string pattern matching algorithm. It is to be proved

```
Name: BackwardLTPM
Input: The subject tree in $pref\_ranked\_bar(subject)$ notation of size $n$
Input: The tree pattern in $pref\_ranked\_bar(pattern)$ notation of size $m$
Input: The subtree jump table $SJT\_bar(subject)$
Input: The bad character shift table $BCS\_pref\_ranked\_bar(pattern)$
Result: Locations of occurrences of tree pattern pattern in subject tree subject
1  begin
2      $i := 0$;
3      while $i <= (n - m)$ do
4          $j := m$;
5          $position := i + j$;
6          while $j > 0$ and $position > 0$ do
7              if $pref\_ranked\_bar(subject)[position] = pref\_ranked\_bar(pattern)[j]$
                 then
8                  $position := position - 1$;
9              else if $pref\_ranked\_bar(pattern)[j] = \uparrow S$ and
                 $pref\_ranked\_bar(subject)[position] \in \mathcal{A}_{\uparrow}$ then
10                 $position := SJT\_bar(subject)[position]$;
11                 $j = j - 1$; {Subtree skip}
12             else break;
13             $j := j - 1$;
14         end
15         if $j = 0$ then yield $position + 1$;
16         $i := i + BCS\_pref\_ranked\_bar(pattern)[subject[i + m]]$;
17     end
18 end
```

Algorithm 24: Backward linearised tree pattern matching algorithm.

that shifting using the $BCS\_pref\_ranked\_bar(p)$ cannot skip any occurrence of the tree pattern p. Let there be a symbol $c = pref\_ranked\_bar(t)[i]$, where i is the position of current match attempt and $c \in \mathcal{A}$. Assume that there is an occurrence of p located at position $i + shift$, $0 < shift < BCS\_pref\_ranked\_bar(p)[c]$. A symbol c must then be located at some position $shift$ either directly or as a part of a subtree that corresponds to a wildcard S. According to Definition 5.2.4, the $BCS\_pref\_ranked\_bar(p)[c]$ is derived from the last occurrence of symbol c in the prefix ranked bar notation of the pattern $pref\_ranked\_bar(p)$, hence we get a contradiction. The shift is also derived from the last occurrence of symbol S and its bar $\uparrow S$ in the prefix ranked bar notation of the pattern $pref\_ranked\_bar(p)$. If the symbol c is located in the subtree that corresponds to a wildcard S and its bar $\uparrow S$, then the shift is already computed from the smallest possible subtree containing the symbol c. Hence, pattern p cannot occur at position $shift$. Therefore, no occurrence of p can be skipped by the algorithm.

The algorithm also checks individual positions inside the subject tree for occurrences.

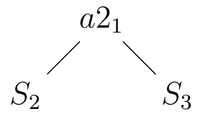


Figure 5.4: Tree pattern p_{10r} from Example 5.2.11.

Table 5.3: A trace of the run of Algorithm 24 (BackwardLTPM) for subject tree t_{4r} and tree pattern p_{10r}.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	
$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$	$pref\_ranked\_bar(t_{4r})$
15	8	5	2	7	4	1	14	11	8	13	10	7	0	$subtree\_jumps(t_{4r})$
					$\uparrow 2$									$\uparrow 0 \neq \uparrow 2$, $shift = 1$
	$a2$	$S \!\rightarrow$	$\leftarrow\!\uparrow S$	$S \!\rightarrow$	$\leftarrow\!\uparrow S$	$\uparrow 2$								match, $shift = 1$
							$\uparrow 2$							$a2 \neq \uparrow 0$, $shift = 5$
							$a2$	$S \!\rightarrow$	$\leftarrow\!\uparrow S$	$S \!\rightarrow$	$\leftarrow\!\uparrow S$	$\uparrow 2$		match, $shift = 1$
$a2$	$S \!\rightarrow$					$\leftarrow\!\uparrow S$	$S \!\rightarrow$					$\leftarrow\!\uparrow S$	$\uparrow 2$	match, $shift = 1$

The check is an extension of the same check performed by the backward string pattern matching algorithm. Individual symbols of the pattern tree other than the wildcard S and its bar $\uparrow S$ are compared with the subject tree in the same manner as the string version of the algorithm does. Wildcard S and its bar $\uparrow S$ are handled using table $SJT\_bar(t)$ which allows skipping subtrees of $pref\_ranked\_bar(t)$. $\square$

Example 5.2.11. Consider tree pattern p_{10r} in the prefix ranked bar notation $pref\_ranked\_bar(p_{10r}) = a2\ S\ \uparrow S\ S\ \uparrow S\ \uparrow 2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow S\}$ depicted in Figure 5.4 and a tree t_{4r} in the prefix ranked bar notation $pref\_ranked\_bar\ (t_{4r}) = a2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ \uparrow 2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0\}$.

The $BCS\_pref\_ranked\_bar(p_{10r})$ abbreviated as BCS contains the following items: $BCS[a3] = 6$, $BCS[a2] = 5$, $BCS[a1] = 4$, $BCS[a0] = 2$, $BCS[\uparrow 3] = 1$, $BCS[\uparrow 2] = 1$, $BCS[\uparrow 1] = 1$, $BCS[\uparrow 0] = 1$.

A trace of the run of Algorithm 24 (BackwardLTPM) is depicted in Table 5.3. Longer subtrees in place of wildcards S are denoted by $S\!\rightarrow\ \leftarrow\!\uparrow S$ $\triangle$.

The trace of the run of the Algorithm 24 (BackwardLTPM) from Example 5.2.11 starts with a pattern aligned as close to the left of the subject as possible, hence at position 6 of the $pref\_ranked\_bar(t_{4r})$. Mismatch of $\uparrow 2$ and $\uparrow 0$ results in a subsequent shift by 1 symbol to align $\uparrow 0$ with the position of the end of the last wildcard S in the $pref\_ranked\_bar(p_{8r})$. The algorithm recognises pattern match on positions 2 to 7 and shifts is by 1 symbol to align $\uparrow 2$ again with the end of the last wildcard S in $pref\_ranked\_bar(p_{8r})$. Mismatch of $\uparrow 2$ and $a2$ results in a shift by 5 symbols where $a2$ from the pattern is aligned with $a2$ from the subject. Another match is recognised and the shift is by 1 symbol is performed, where another and last occurrence is recognised and the subsequent shift is to the outside of the $pref\_ranked\_bar(t_{4r})$ resulting in the termination of the run of the Algorithm 24 (BackwardLTPM) and return of the set of occurrences.

Lengths of shifts strongly depend on the position of the symbol S in the pattern. Shifts are longer with increasing the distance of the symbol S from the end of the pattern.

The bad character shift table is the only precomputed data structure from the pattern needed for the backward linearised tree pattern algorithm and its size is $\Theta(|\mathcal{A}|)$, where $|\mathcal{A}|$ is the alphabet size. The preprocessing time is $O(m + |\mathcal{A}|)$, where m is the pattern length and $|\mathcal{A}|$ is the alphabet size.

Backward string pattern matching is known to perform the sublinear number of comparisons of symbols on average. The modification to backward linearised tree pattern matching requires the subject tree to be read in the prefix ranked bar notation. However, the algorithm still performs $\Omega(\frac{n}{m})$ comparisons of symbols, where n is the size of the subject tree and m is the size of the given tree pattern and $O(n * m)$ comparisons of symbols as in the case of the backward string pattern matching. The lengths of the shifts depend on the position of the last wildcard S in pattern p – the closer to the end of the pattern the last occurrence of symbol S is, the longer shifts are performed.

5.3 Reversed variant

The obvious relation between the length of the shift and the last occurrence of symbol S may be addressed in some special case with the introduction of a reversed variant of the backward linearised tree pattern matching algorithm. The reverse is in the meaning of reversing the shifting of the pattern and simultaneously reversing the direction of symbol comparison in the match attempt. The prefix ranked bar notation from Definition 5.2.1 is used and the subtree jump table for bar notation from Definition 5.2.7 is also used without changes. The definition of bad character shift must be changed accordingly to represent shifts in other direction. The content of this section is novel and not presented in any publication.

Definition 5.3.1. Let $pattern[1..m]$ be a $pref\_ranked\_bar$ notation of a tree pattern p over an alphabet $\mathcal{A}$. The *reversed bad character shift table* $RBCS\_pref\_ranked\_bar(p)$ for backward linearised tree pattern matching is defined for each $a \in \mathcal{A}$.

$$RBCS\_pref\_ranked\_bar(p)[a] =$$
$$\min \left(\begin{array}{l} \{m\} \cup \{j-1 : pattern[j] = a \text{ and } m \geq j > 1\} \cup \\ \{j-1+Arity(a)*2 : pattern[j] = \uparrow S \text{ and } m \geq j > 1 \text{ and } a \in \mathcal{A}_{\uparrow}\} \cup \\ \{j-2 : pattern[j] = \uparrow S \text{ and } m \geq j > 1 \text{ and } a \notin \mathcal{A}_{\uparrow} \text{ and } pattern \neq S \uparrow S\} \cup \\ \{1 : a \notin \mathcal{A}_{\uparrow} \text{ and } pattern = S \uparrow S\} \end{array} \right)$$

The idea behind Definition 5.3.1 is similar to one behind Definition 5.2.4. The third, fourth and fifth part of the union in the definition is changed to reflect a change in the direction of the shift. The change in the direction of the shift corresponds to a change in the direction of processing symbols of the prefix ranked bar representation of the pattern. The change in the direction of processing symbols of the representation of the pattern causes the change in the order of visits of bar symbols and non-bar symbols of individual subtrees. The same change happens for symbols S and $\uparrow S$. This symmetry is translated to the third, fourth and fifth part of the union in the definition so that the $\uparrow S$ is searched to prolong the shift for bar symbol by their arity.

```
Name: ConstructRBCS
Input: The tree pattern in the prefix ranked bar notation
       $pref\_ranked\_bar(pattern)$ of size $m$ over an alphabet $\mathcal{A}$ of the subject
       tree
Result: The bad character shift table $RBCS\_pref\_ranked\_bar(pattern)$
begin
    $s := m$;
    for $i := m$ downto 1 do
        if $pref\_ranked\_bar(pattern)[i] = \uparrow S$ then $s := i - 1$;
    end
    foreach $x \in \mathcal{A}$ do $RBCS\_pref\_ranked\_bar(pattern)[x] := m$;
    foreach $x \in \mathcal{A}$ do
        if $x \in \mathcal{A}_{\uparrow}$ then $shift := s + Arity(x) * 2$;
        else if $s >= 2$ then $shift := s - 1$;
        else $shift := s$;
        if $RBCS\_pref\_ranked\_bar(pattern)[x] > shift$ then
            $RBCS\_pref\_ranked\_bar(pattern)[x] := shift$;
        end
    end
    for $i := m$ downto 2 do
        if $pref\_ranked\_bar(pattern)[i] \notin \{S, \uparrow S\}$ and
         $RBCS\_pref\_ranked\_bar(pattern)[pref\_ranked\_bar(pattern)[i]] > (i - 1)$
         then
            $RBCS\_pref\_ranked\_bar(pattern)[pref\_ranked\_bar(pattern)[i]] :=$
              $i - 1$;
        end
    end
    return $RBCS\_pref\_ranked\_bar(pattern)$;
end
```

Algorithm 25: Construction of RBCS table.

Algorithm 25 (ConstructRBCS) for construction of $RBCS(pref\_ranked\_bar(pattern))$ for reversed backward linearised tree pattern matching has been modified in the same way as the Definition 5.3.1 of reversed bad character shift table.

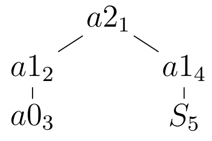


Figure 5.5: Tree pattern p_{11r} from Example 5.3.3.

Example 5.3.2. Consider tree pattern p_{8r}, depicted in Figure 5.1, in the prefix ranked bar notation $pref\_ranked\_bar(p_{8r}) = a2\ a1\ S\ {\uparrow}S\ {\uparrow}1\ a1\ a0\ {\uparrow}0\ {\uparrow}1\ {\uparrow}2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, {\uparrow}3, {\uparrow}2, {\uparrow}1, {\uparrow}0, {\uparrow}S\}$. Algorithm 25 (ConstructRBCS) constructs the following items of the reversed bad character shift table $RBCS\_pref\_ranked\_bar(p_{8r})$ abbreviated as $RBCS$. △

$$\begin{aligned}
RBCS[a3] &= \min(\{10\} \cup \emptyset \cup \{2\}) = 2 & RBCS[a2] &= \min(\{10\} \cup \emptyset \cup \{2\}) = 2\\
RBCS[a1] &= \min(\{10\} \cup \{1,5\} \cup \{2\}) = 1 & RBCS[a0] &= \min(\{10\} \cup \{6\} \cup \{2\}) = 2\\
RBCS[{\uparrow}3] &= \min(\{10\} \cup \emptyset \cup \{9\}) = 9 & RBCS[{\uparrow}2] &= \min(\{10\} \cup \{9\} \cup \{7\}) = 7\\
RBCS[{\uparrow}1] &= \min(\{10\} \cup \{4,8\} \cup \{5\}) = 4 & RBCS[{\uparrow}0] &= \min(\{10\} \cup \{7\} \cup \{3\}) = 3
\end{aligned}$$

Tree pattern p_{8r} used in Example 5.3.2 is the same as in Example 5.2.5 and the lengths of shifts are shorter which is purely caused by the particular selection of the pattern.

Example 5.3.3. However, consider tree pattern p_{11r}, depicted in Figure 5.5, in the prefix ranked bar notation $pref\_ranked\_bar(p_{11r}) = a2\ a1\ a0\ {\uparrow}0\ {\uparrow}1\ a1\ S\ {\uparrow}S\ {\uparrow}1\ {\uparrow}2$ over alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, {\uparrow}3, {\uparrow}2, {\uparrow}1, {\uparrow}0, {\uparrow}S\}$. Algorithm 22 (ConstructBCS_pref_ranked_bar) constructs the following items of the bad character shift table $BCS\_pref\_ranked\_bar(p_{11r})$ abbreviated as BCS.

$$\begin{aligned}
BCS[a3] &= \min(\{10\} \cup \emptyset \cup \{9\}) = 9 & BCS[a2] &= \min(\{10\} \cup \{9\} \cup \{7\}) = 7\\
BCS[a1] &= \min(\{10\} \cup \{4,8\} \cup \{5\}) = 4 & BCS[a0] &= \min(\{10\} \cup \{7\} \cup \{3\}) = 3\\
BCS[{\uparrow}3] &= \min(\{10\} \cup \emptyset \cup \{2\}) = 2 & BCS[{\uparrow}2] &= \min(\{10\} \cup \emptyset \cup \{2\}) = 2\\
BCS[{\uparrow}1] &= \min(\{10\} \cup \{1,5\} \cup \{2\}) = 1 & BCS[{\uparrow}0] &= \min(\{10\} \cup \{6\} \cup \{2\}) = 2
\end{aligned}$$

And Algorithm 25 (ConstructRBCS) constructs the following items of the reversed bad character shift table $RBCS\_pref\_ranked\_bar(p_{11r})$ abbreviated as $RBCS$. △

$$\begin{aligned}
RBCS[a3] &= \min(\{10\} \cup \emptyset \cup \{6\}) = 6 & RBCS[a2] &= \min(\{10\} \cup \emptyset \cup \{6\}) = 6\\
RBCS[a1] &= \min(\{10\} \cup \{1,5\} \cup \{6\}) = 1 & RBCS[a0] &= \min(\{10\} \cup \{2\} \cup \{6\}) = 2\\
RBCS[{\uparrow}3] &= \min(\{10\} \cup \emptyset \cup \{13\}) = 10 & RBCS[{\uparrow}2] &= \min(\{10\} \cup \{9\} \cup \{11\}) = 9\\
RBCS[{\uparrow}1] &= \min(\{10\} \cup \{4,8\} \cup \{9\}) = 4 & RBCS[{\uparrow}0] &= \min(\{10\} \cup \{3\} \cup \{7\}) = 3
\end{aligned}$$

Tree pattern p_{11r} is chosen to show the property of the reversed bad character shift table in Example 5.3.3. The shifts by the reversed bad character shift table are longer than by the bad character shift table for this particular pattern.

```
Name: ReversedBackwardLTPM
Input: The subject tree in the prefix ranked bar notation
       $pref\_ranked\_bar(subject)$ of size $n$
Input: The subtree jump table for bar notation $SJT\_bar(subject)$
Input: The tree pattern in the prefix ranked bar notation
       $pref\_ranked\_bar(pattern)$ of size $m$
Input: The reversed bad character shift table $RBCS\_pref\_ranked\_bar(pattern)$
Result: Locations of occurrences of tree pattern $pattern$ in subject tree $subject$
begin
    $i := n - m + 1$;
    while $i >= 1$ do
        $j = 1$;
        $position = i$;
        while $j <= m$ and $position <= n$ do
            if $pref\_ranked\_bar(subject)[position] = pref\_ranked\_bar(pattern)[j]$ then
                $position := position + 1$;
            else if $pref\_ranked\_bar(pattern)[j] = S$ and
             $pref\_ranked\_bar(subject)[position] \notin \mathcal{A}_{\uparrow}$ then
                $position := SJT\_bar(subject)[position]$;
                $j := j + 1$;
            else break;
            $j := j + 1$;
        end
        if $j = m + 1$ then yield $i$;
        $i := i - RBCS\_pref\_ranked\_bar[pref\_ranked\_bar(subject)[i]]$;
    end
end
```

Algorithm 26: Reversed backward linearised tree pattern matching algorithm.

Algorithm 26 (ReversedBackwardLTPM) is similar to Algorithm 24 (BackwardLTPM). The algorithm is modified to work in reverse when compared to Algorithm 24 (BackwardLTPM) introduced in the previous section. It starts searching with the pattern (in the prefix ranked bar notation) aligned to the right end of the subject tree (also in the prefix ranked bar notation). The pattern is shifted towards the beginning of the subject. Symbols of the pattern and the subject on a particular position are compared from right to left.

Theorem 5.3.4. Given tree pattern p in the prefix ranked bar notation and reversed bad character shift table $RBCS\_pref\_ranked\_bar(p)$ constructed for the tree pattern p by Algorithm 25 (ConstructRBCS), Algorithm 26 (ReversedBackwardLTPM) correctly computes the locations of all occurrences of tree pattern p in subject tree t.

Table 5.4: A trace of the run of Algorithm 26 (ReversedBackwardLTPM) for subject tree t_{4r} and tree pattern p_{10r}.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	
$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$	$pref\_ranked\_bar(t_{4r})$
15	8	5	2	7	4	1	14	11	8	13	10	7	0	$subtree\_jumps(t_{4r})$
								$a2$						$a0 \neq a2$, $shift = 1$
							$a2$	$S{\rightarrow}$	$\leftarrow S\uparrow$	$S{\rightarrow}$	$\leftarrow S\uparrow$	$\uparrow 2$		match, $shift = 1$
						$a2$								$\uparrow 2 \neq a2$, $shift = 5$
	$a2$	$S{\rightarrow}$	$\leftarrow S\uparrow$	$S{\rightarrow}$	$\leftarrow S\uparrow$	$\uparrow 2$								match, $shift = 1$
$a2$	$S{\rightarrow}$					$\leftarrow S\uparrow$	$S{\rightarrow}$					$\leftarrow S\uparrow$	$\uparrow 2$	match, $shift = 1$

Proof. The proof is similar to the proof of Theorem 5.2.10 and it is presented in an appendix as proof of Theorem A.1.1. □

Example 5.3.5. Consider tree pattern p_{10r} in the prefix ranked bar notation $pref\_ranked\_bar(p_{10r}) = a2\ S \uparrow S\ S \uparrow S \uparrow 2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow S\}$ depicted in Figure 5.4 and a tree t_{4r} in the prefix ranked bar notation $pref\_ranked\_bar(t_{4r}) = a2\ a2\ a0 \uparrow 0\ a0 \uparrow 0 \uparrow 2\ a2\ a0 \uparrow 0\ a0 \uparrow 0 \uparrow 2 \uparrow 2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0\}$.

The $RBCS\_pref\_ranked\_bar(p_{10r})$ abbreviated as $RBCS$ contains the following items: $RBCS[a3] = 1$, $RBCS[a2] = 1$, $RBCS[a1] = 1$, $RBCS[a0] = 1$, $RBCS[\uparrow 3] = 6$, $RBCS[\uparrow 2] = 5$, $RBCS[\uparrow 1] = 4$, $RBCS[\uparrow 0] = 2$.

A trace of the run of Algorithm 26 (ReversedBackwardLTPM) is depicted in Table 5.4. Longer subtrees in place of wildcards S are denoted by $S{\rightarrow}\ {\leftarrow}{\uparrow}S$. △

The trace of the run of Algorithm 26 (ReversedBackwardLTPM) from Example 5.3.5 is symmetric to the trace of the run of Algorithm 24 (BackwardLTPM) from Example 5.2.11. The symmetricity is caused by the symmetricity of both the pattern tree and the subject tree. The length of the shift depends on the position of the symbol S in the pattern. Shifts are longer with the distance of the symbol S to the beginning of the pattern.

Since the modification is only in the direction of shifting and symbol comparison, the algorithm properties stay the same. The space complexity of the algorithm $\Theta(|\mathcal{A}|)$ is given by the size of the reversed bad character shift table. The preprocessing time is $O(m+|\mathcal{A}|)$, where m is the pattern length and $|\mathcal{A}|$ is the alphabet size.

The two so far presented variants of the algorithm expect the tree in prefix ranked bar notation. An observation on the structure of the tree represented in prefix ranked bar notations is that the diversity of bars depend solely on the number of different arities, whereas two non-bar symbols can differ additionally by their label. It is expected that the actual execution performance will depend not only on the lengths of shifts but also on the ability to early detect failed occurrence check of a given tree pattern on some concrete position within the subject tree. This ability leads to an expectation that the reversed

Table 5.5: Prefix, postfix, prefix bar, postfix bar, prefix ranked bar, and postfix ranked bar linear notations of tree t_{1r}.

prefix_bar	a	a	a	$\uparrow$	a	a	$\uparrow$	$\uparrow$	$\uparrow$	a	a	$\uparrow$	$\uparrow$	$\uparrow$
postfix_bar	$\uparrow$	$\uparrow$	$\uparrow$	a	$\uparrow$	$\uparrow$	a	a	a	$\uparrow$	$\uparrow$	a	a	a
prefix	$a2$	$a2$	$a0$		$a1$	$a0$				$a1$	$a0$			
postfix				$a0$			$a0$	$a1$	$a2$			$a0$	$a1$	$a2$
prefix_ranked_bar	$a2$	$a2$	$a0$	$\uparrow 0$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$
postfix_ranked_bar	$\uparrow 2$	$\uparrow 2$	$\uparrow 0$	$a0$	$\uparrow 1$	$\uparrow 0$	$a0$	$a1$	$a2$	$\uparrow 1$	$\uparrow 0$	$a0$	$a1$	$a2$

variant of the algorithm should benefit from the prefix ranked bar notation as non-bar symbols are tested sooner than bar symbols.

5.4 Postfix variant

The selection of the prefix ranked bar notation for the backward linearised tree pattern matching algorithm is not strict. One can choose a postfix ranked bar notation and modify the backward linearised tree pattern matching algorithm to work with it. The definition of the bad character shift must be changed for the postfix ranked bar notation as well. However, the subtree jump table for bar notation from Definition 5.2.7 can be used without changes. The content of this section is novel and not presented in any publication.

Definition 5.4.1. The *postfix ranked bar notation* $post\_ranked\_bar(t)$ of a tree t is defined as follows:

1. $post\_ranked\_bar(S) = \uparrow S\ S$
2. $post\_ranked\_bar(a) = \uparrow 0\ a0$ if a is a leaf,
3. $post\_ranked\_bar(t) = \uparrow n\ post\_ranked\_bar(b_1) \dots post\_ranked\_bar(b_n)\ an$, where a is the root of the tree t, $n = Arity(a)$ and $b_1, \dots\ b_n$ are direct descendants of a.

The postfix ranked bar notation is a combination of postfix notation and postfix bar notation similarly as in case of prefix ranked bar notation defined in Definition 5.2.1. The properties of postfix ranked bar notation is the same as properties of prefix ranked bar notation.

Example 5.4.2. Consider tree t_{1r} and its prefix, postfix, prefix bar, postfix bar, prefix ranked bar, and postfix ranked bar notations. The linear notations of tree t_{1r} are all together illustrated in Table 5.5. Table 5.5 is an extension of Table 5.1 from Example 5.2.2.
$\triangle$

The base variant of backward linearised tree pattern matching algorithm working on postfix ranked bar notation will shift the pattern from right to left. Hence the algorithms and definitions are going to be similar to the ones defined in Section 5.3.

Definition 5.4.3. Let $pattern[1..m]$ be a *post_ranked_bar* notation of tree pattern p over alphabet $\mathcal{A}$. The *postfix bad character shift table* $BCS\_post\_ranked\_bar(p)$ for the backward linearised tree pattern matching is defined for each $a \in \mathcal{A}$.

$$BCS\_post\_ranked\_bar(p)[a] =$$

$$\min\left(\begin{array}{l} \{m\} \cup \{j-1 : pattern[j] = a \text{ and } m \geq j > 1\} \cup \\ \{j-1+Arity(a)*2 : pattern[j] = S \text{ and } m \geq j > 1 \text{ and } a \notin \mathcal{A}_{\uparrow}\} \cup \\ \{j-2 : pattern[j] = S \text{ and } m \geq j > 1 \text{ and } a \in \mathcal{A}_{\uparrow} \text{ and } pattern \neq {\uparrow}S\ S\} \cup \\ \{1 : a \in \mathcal{A}_{\uparrow} \text{ and } pattern = {\uparrow}S\ S\} \end{array}\right)$$

Definition 5.4.3 is similar to Definition 5.3.1. The direction of shifts the same, however, the definition is changed to reflect change in the used notation – S is searched instead of $\uparrow S$ and the length of shifts for non-bar symbols considered to be inside the $\uparrow S\ S$ are prolonged.. The computation of shifts is otherwise the same.

Algorithm 27 (ConstructPBCS) for construction of $BCS\_post\_ranked\_bar(pattern)$ for the postfix backward linearised tree pattern matching has been modified in the same way as the definition of postfix bad character shift table from Definition 5.4.3.

Example 5.4.4. Consider tree pattern p_{8r}, depicted in Figure 5.1, in the postfix ranked bar notation $post\_ranked\_bar(p_{8r}) = {\uparrow}2\ {\uparrow}1\ {\uparrow}S\ S\ a1\ {\uparrow}1\ {\uparrow}0\ a0\ a1\ a2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, {\uparrow}3, {\uparrow}2, {\uparrow}1, {\uparrow}0, {\uparrow}S\}$. Algorithm 27 (ConstructPBCS) constructs the following items of the postfix bad character shift table $BCS\_post\_ranked\_bar(p_{8r})$ abbreviated as $PBCS$. △

$PBCS[a3] = \min(\{10\} \cup \emptyset \cup \{9\}) = 9$	$PBCS[a2] = \min(\{10\} \cup \{9\} \cup \{7\}) = 7$
$PBCS[a1] = \min(\{10\} \cup \{4,8\} \cup \{5\}) = 4$	$PBCS[a0] = \min(\{10\} \cup \{7\} \cup \{3\}) = 3$
$PBCS[{\uparrow}3] = \min(\{10\} \cup \emptyset \cup \{2\}) = 2$	$PBCS[{\uparrow}2] = \min(\{10\} \cup \emptyset \cup \{2\}) = 2$
$PBCS[{\uparrow}1] = \min(\{10\} \cup \{1,5\} \cup \{2\}) = 1$	$PBCS[{\uparrow}0] = \min(\{10\} \cup \{6\} \cup \{2\}) = 2$

Tree pattern p_{8r} used in Example 5.4.4 is the same as in Example 5.3.2 and the lengths of shifts for bar symbols and terminal symbols are swapped. This is obvious when Definition 5.2.1 is compared with Definition 5.4.1. Basically bar symbols and non-bar symbols are swapped in the prefix ranked bar notation and postfix ranked bar notation.

Algorithm 28 (PostfixBackwardLTPM) is similar to Algorithm 26 (ReversedBackward-LTPM). It is modified to work with the postfix ranked bar notation of a tree. The direction of shifting of the pattern and the direction of symbol comparison is borrowed from the reversed backward linearised tree pattern matching algorithm. Therefore, it starts searching with pattern (in the postfix ranked bar notation) aligned to the right end of the subject tree (also in the postfix ranked bar notation), and pattern is shifted towards the beginning of the subject whereas the symbols of the pattern and the subject on a particular position are compared from left to right.

```
Name: ConstructPBCS
Input: The tree pattern in the postfix ranked bar notation
       $post\_ranked\_bar(pattern)$ of size $m$ over an alphabet $\mathcal{A}$ of the subject tree
Result: The bad character shift table $BCS\_post\_ranked\_bar(pattern)$
begin
    $s := m$;
    for $i := m$ downto 1 do
        if $post\_ranked\_bar(pattern)[i] = S$ then $s := i - 1$;
    end
    foreach $x \in \mathcal{A}$ do $BCS\_post\_ranked\_bar(pattern)[x] := m$;
    foreach $x \in \mathcal{A}$ do
        if $x \notin \mathcal{A}_{\uparrow}$ then $shift := s + Arity(x) * 2$;
        else if $s >= 2$ then $shift := s - 1$;
        else $shift := s$;
        if $BCS\_post\_ranked\_bar(pattern)[x] > shift$ then
            $BCS\_post\_ranked\_bar(pattern)[x] := shift$;
        end
    end
    for $i := m$ downto 2 do
        if $post\_ranked\_bar(pattern)[i] \notin \{S, \uparrow S\}$ and
          $BCS\_post\_ranked\_bar(pattern)[post\_ranked\_bar(pattern)[i]] > (i - 1)$
          then
            $BCS\_post\_ranked\_bar(pattern)[post\_ranked\_bar(pattern)[i]] := i - 1$;
        end
    end
    return $BCS\_post\_ranked\_bar(pattern)$;
end
```

Algorithm 27: Construction of BCS_post_ranked_bar table.

Theorem 5.4.5. Given tree pattern p in the postfix ranked bar notation and postfix bad character shift table $BCS\_post\_ranked\_bar(p)$ constructed for the tree pattern p by Algorithm 27 (ConstructPBCS), Algorithm 28 (PostfixBackwardLTPM) correctly computes the locations of all occurrences of tree pattern p in subject tree t.

Proof. The proof is similar to the proof of Theorem 5.2.10 and it is presented in an appendix as proof of Theorem A.1.2. ☐

Example 5.4.6. Consider tree pattern p_{10r} in the postfix ranked bar notation $post\_ranked\_bar(p_{10r}) = \uparrow 2 \uparrow S\ S \uparrow S\ S\ a2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow S\}$ depicted in Figure 5.4 and a tree t_{4r} in the postfix ranked bar notation $post\_ranked\_bar(t_{4r}) = \uparrow 2 \uparrow 2 \uparrow 0\ a0 \uparrow 0\ a0\ a2 \uparrow 2 \uparrow 0\ a0 \uparrow 0\ a0\ a2\ a2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0\}$.

```
Name: PostfixBackwardLTPM
Input: The subject tree in the postfix ranked bar notation
       $post\_ranked\_bar(subject)$ of size $n$
Input: The subtree jump table for bar notation $SJT\_bar(subject)$
Input: The tree pattern in the postfix ranked bar notation
       $post\_ranked\_bar(pattern)$ of size $m$
Input: The bad character shift table $BCS\_post\_ranked\_bar(pattern)$
Result: Locations of occurrences of tree pattern $pattern$ in subject tree $subject$
begin
    $i := n - m + 1$;
    while $i >= 1$ do
        $j = 1$;
        $position = i$;
        while $j <= m$ and $position <= n$ do
            if $post\_ranked\_bar(subject)[position] = post\_ranked\_bar(pattern)[j]$
             then
                $position := position + 1$;
            else if $post\_ranked\_bar(pattern)[j] = \uparrow S$ and
             $post\_ranked\_bar(subject)[position] \in \mathcal{A}_{\uparrow}$ then
                $position := SJT\_bar(subject)[position]$;
                $j := j + 1$;
            else break;
            $j := j + 1$;
        end
        if $j = m + 1$ then yield $i$;
        $i := i - BCS\_post\_ranked\_bar(pattern)[post\_ranked\_bar(subject)[i]]$;
    end
end
```

Algorithm 28: Postfix backward linearised tree pattern matching algorithm.

The $BCS\_post\_ranked\_bar(p_{10r})$ abbreviated as $PBCS$ contains the following items: $PBCS[a3] = 6$, $PBCS[a2] = 5$, $PBCS[a1] = 4$, $PBCS[a0] = 2$, $PBCS[\uparrow 3] = 1$, $PBCS[\uparrow 2] = 1$, $PBCS[\uparrow 1] = 1$, $PBCS[\uparrow 0] = 1$.

A trace of the run of the Algorithm 28 (PostfixBackwardLTPM) is depicted in Table 5.6. Longer subtrees in place of wildcards S are denoted by $\uparrow S \rightarrow\ \leftarrow S$. △

The trace of the run of Algorithm 28 (PostfixBackwardLTPM) from Example 5.4.6 is similar to the trace of the run of Algorithm 26 (ReversedBackwardLTPM) from Example 5.3.5. The similarity is caused by the similarity in the notations used – prefix and postfix ranked bar notations. The length of the shift depends on the position of the symbol S in the pattern in the same way as in case of the reversed backward linearised tree pattern matching algorithm.

Table 5.6: A trace of the run of Algorithm 28 (PostfixBackwardLTPM) for subject tree t_{4r} and tree pattern p_{10r}.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	
$\uparrow 2$	$\uparrow 2$	$\uparrow 0$	$a0$	$\uparrow 0$	$a0$	$a2$	$\uparrow 2$	$\uparrow 0$	$a0$	$\uparrow 0$	$a0$	$a2$	$a2$	$post\_ranked\_bar(t_{4r})$
15	8	5	2	7	4	1	14	11	8	13	10	7	0	$subtree\_jumps(t_{4r})$
								$\uparrow 2$						$\uparrow 0 \neq \uparrow 2$, $shift = 1$
							$\uparrow 2$	$\uparrow S \rightarrow$	$\leftarrow S$	$\uparrow S \rightarrow$	$\leftarrow S$	$a2$		match, $shift = 1$
						$\uparrow 2$								$a2 \neq \uparrow 2$, $shift = 5$
	$\uparrow 2$	$\uparrow S \rightarrow$	$\leftarrow S$	$\uparrow S \rightarrow$	$\leftarrow S$	$a2$								match, $shift = 1$
$\uparrow 2$	$\uparrow S \rightarrow$					$\leftarrow S$	$\uparrow S \rightarrow$					$\leftarrow S$	$a2$	match, $shift = 1$

Since the modification is only in the used ranked bar notation the algorithm properties stay the same. The space complexity of the algorithm $\Theta(|\mathcal{A}|)$ is given by the size of the postfix bad character shift table. The preprocessing time is $O(m + |\mathcal{A}|)$, where m is the pattern length and $|\mathcal{A}|$ is the alphabet size.

5.5 Reversed postfix variant

It is possible to introduce a postfix reversed variant of linearised tree pattern matching algorithm similarly as a reversed variant of backward linearised tree pattern matching was introduced in Section 5.3. Hence the directions of shifting and symbol comparison used in the postfix variant can be reversed. The postfix ranked bar notation from Definition 5.4.1 and the subtree jump table for bar notation from Definition 5.2.7 can be used without changes. However, the definition of bad character shift must be changed. The content of this section is novel and not presented in any publication.

Definition 5.5.1. Let $pattern[1..m]$ be a postfix ranked bar notation of tree pattern p over alphabet $\mathcal{A}$. The *postfix reversed bad character shift table* $RBCS\_post\_ranked\_bar(p)$ for the backward linearised tree pattern matching is defined for each $a \in \mathcal{A}$.

$$RBCS\_post\_ranked\_bar(p)[a] =$$
$$\min \left(\begin{array}{l} \{m\} \cup \{j : pattern[m-j] = a \text{ and } m > j > 0\} \cup \\ \{j + Arity(a) * 2 : pattern[m-j] = \uparrow S \text{ and } m > j > 0 \text{ and } a \in \mathcal{A}_\uparrow\} \cup \\ \{j - 1 : pattern[m-j] = \uparrow S \text{ and } m > j > 0 \text{ and } a \notin \mathcal{A}_\uparrow \text{ and } pattern \neq \uparrow S\ S\} \cup \\ \{1 : a \notin \mathcal{A}_\uparrow \text{ and } pattern = \uparrow S\ S\} \end{array} \right)$$

Definition 5.5.1 is similar to Definition 5.2.4. The direction of shifts the same, however, the definition is changed to reflect the change in the used notation – $\uparrow S$ is searched instead of S and the length of shifts for bar symbols considered to be inside the $\uparrow S\ S$ are prolonged. The computation of shifts is otherwise the same.

```
Name: ConstructPRBCS
Input: Tree $pattern$ in the postfix ranked bar notation $post\_ranked\_bar(pattern)$
       of size $m$ over alphabet $\mathcal{A}$ of the $subject$ tree
Result: The postfix reversed bad character shift table
        $RBCS\_post\_ranked\_bar(pattern)$
begin
    $s := m$;
    for $i := 1$ to $m$ do
        if $post\_ranked\_bar(pattern)[i] = \uparrow S$ then $s = m - i$;
    end
    foreach $x \in \mathcal{A}$ do $RBCS\_post\_ranked\_bar(pattern)[x] = m$;
    foreach $x \in \mathcal{A}$ do
        if $x \in \mathcal{A}_{\uparrow}$ then $shift := s + Arity(x) * 2$;
        else if $s >= 2$ then $shift := s - 1$;
        else $shift := s$;
        if $RBCS\_post\_ranked\_bar(pattern)[x] > shift$ then
            $RBCS\_post\_ranked\_bar(pattern)[x] := shift$;
        end
    end
    for $i := 1$ to $m - 1$ do
        if $post\_ranked\_bar(pattern)[i] \notin \{S, \uparrow S\}$ and
         $RBCS\_post\_ranked\_bar(pattern)[post\_ranked\_bar(pattern)[i]] > (m - i)$
         then
            $RBCS\_post\_ranked\_bar(pattern)[post\_ranked\_bar(pattern)[i]] := m - i$;
        end
    end
    return $RBCS\_post\_ranked\_bar(pattern)$;
end
```

Algorithm 29: Construction of RBCS_post_ranked_bar table.

Algorithm 29 (ConstructPRBCS) of construction of $RBCS\_post\_ranked\_bar(pattern)$ for postfix reversed backward linearised tree pattern matching has been modified in the same way as the definition of the postfix reversed bad character shift table 5.4.3.

Example 5.5.2. Consider tree pattern p_{8r}, depicted in Figure 5.1, in the postfix ranked bar notation $post\_ranked\_bar(p_{8r}) = \uparrow 2\ \uparrow 1\ \uparrow S\ S\ a1\ \uparrow 1\ \uparrow 0\ a0\ a1\ a2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow S\}$. Algorithm 29 (ConstructPRBCS) constructs the following items of the postfix reversed bad character shift table $RBCS\_post\_ranked\_bar(p_{8r})$ abbreviated as $RBCS$. △

$RBCS[a3] = \min(\{10\} \cup \emptyset \cup \{6\}) = 6$	$RBCS[a2] = \min(\{10\} \cup \emptyset \cup \{6\}) = 6$
$RBCS[a1] = \min(\{10\} \cup \{1,5\} \cup \{6\}) = 1$	$RBCS[a0] = \min(\{10\} \cup \{2\} \cup \{6\}) = 2$
$RBCS[\uparrow 3] = \min(\{10\} \cup \emptyset \cup \{13\}) = 10$	$RBCS[\uparrow 2] = \min(\{10\} \cup \{9\} \cup \{11\}) = 9$
$RBCS[\uparrow 1] = \min(\{10\} \cup \{4,8\} \cup \{9\}) = 4$	$RBCS[\uparrow 0] = \min(\{10\} \cup \{3\} \cup \{7\}) = 3$

Tree pattern p_{8r} used in Example 5.5.2 is the same as in Example 5.2.5. The lengths of shifts for bar symbols and terminal symbols are swapped similarly as in Example 5.4.4.

```
Name: PostfixReversedBackwardLTPM
Input: The subject tree in the postfix ranked bar notation
       $post\_ranked\_bar(subject)$ of size $n$
Input: The subtree jump table for bar notation $SJT\_bar(subject)$
Input: The tree pattern in the postfix ranked bar notation
       $post\_ranked\_bar(pattern)$ of size $m$
Input: The postfix reversed bad character shift table
       $RBCS\_post\_ranked\_bar(pattern)$
Result: Locations of occurrences of tree pattern $pattern$ in subject tree $subject$
1  begin
2      $i := 0$;
3      while $i <= (n - m)$ do
4          $j := m$;
5          $position := i + j$;
6          while $j > 0$ and $position > 0$ do
7              if $post\_ranked\_bar(subject)[position] = post\_ranked\_bar(pattern)[j]$
                then
8                  $position := position - 1$;
9              else if $post\_ranked\_bar(pattern)[j] = S$ and
                $post\_ranked\_bar(subject)[position] \notin \mathcal{A}_\uparrow$ then
10                 $position := SJT\_bar(subject)[position]$;
11                 $j = j - 1$; {Subtree skip}
12             else break;
13             $j := j - 1$;
14         end
15         if $j = 0$ then yield $position + 1$;
16         $i := i + RBCS\_post\_ranked\_bar(pattern)[post\_ranked\_bar(subject)[i + m]]$;
17     end
18 end
```

Algorithm 30: Postfix reversed backward linearised tree pattern matching algorithm.

Algorithm 30 (PostfixReversedBackwardLTPM) is similar to Algorithm 28 (Postfix-BackwardLTPM). The algorithm is modified to work in reverse when compared to Algorithm 28 (PostfixBackwardLTPM) introduced in the previous section. But it is also

Table 5.7: A trace of the run of Algorithm 30 (PostfixReversedBackwardLTPM) for subject tree t_{4r} and tree pattern p_{10r}.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	
$\uparrow 2$	$\uparrow 2$	$\uparrow 0$	$a0$	$\uparrow 0$	$a0$	$a2$	$\uparrow 2$	$\uparrow 0$	$a0$	$\uparrow 0$	$a0$	$a2$	$a2$	$post\_ranked\_bar(t_{4r})$
15	8	5	2	7	4	1	14	11	8	13	10	7	0	$subtree\_jumps(t_{4r})$
					$a2$									$a0 \neq a2$, $shift = 1$
	$\uparrow 2$	$\uparrow S \rightarrow$	$\leftarrow S$	$\uparrow S \rightarrow$	$\leftarrow S$	$a2$								match, $shift = 1$
							$a2$							$\uparrow 2 \neq a2$, $shift = 5$
							$\uparrow 2$	$\uparrow S \rightarrow$	$\leftarrow S$	$\uparrow S \rightarrow$	$\leftarrow S$	$a2$		match, $shift = 1$
$\uparrow 2$	$\uparrow S \rightarrow$					$\leftarrow S$	$\uparrow S \rightarrow$					$\leftarrow S$	$a2$	match, $shift = 1$

similar to Algorithm 24 (BackwardLTPM) where the complementarity of bar and non-bar symbols in prefix and postfix ranked bar notations is the cause of the similarity of Algorithm 30 (PostfixReversedBackwardLTPM) and Algorithm 24 (BackwardLTPM). The direction of shifting of the pattern and the direction of symbol comparison is borrowed from the linearised tree pattern matching algorithm. It starts searching with the pattern (in the prefix ranked bar notation) aligned to the left end of the subject tree (also in the prefix ranked bar notation). The pattern is shifted towards the end of the subject. Symbols of the pattern and the subject on a particular position are compared from right to left.

Theorem 5.5.3. Given tree pattern p in the postfix ranked bar notation and the postfix reversed bad character shift table $RBCS\_post\_ranked\_bar(p)$ constructed for the tree pattern p by Algorithm 29 (ConstructPRBCS), Algorithm 30 (PostfixReversedBackward-LTPM) correctly computes the locations of all occurrences of tree pattern p in subject tree t.

Proof. The proof is similar to the proof of Theorem 5.2.10 and it is presented in an appendix as proof of Theorem A.1.3. ☐

Example 5.5.4. Consider tree pattern p_{10r} in the postfix ranked bar notation $post\_ranked\_bar(p_{10r}) = \uparrow 2 \uparrow S\ S \uparrow S\ S\ a2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow S\}$ depicted in Figure 5.4 and a tree t_{4r} in the postfix ranked bar notation $post\_ranked\_bar(t_{4r}) = \uparrow 2 \uparrow 2 \uparrow 0\ a0 \uparrow 0\ a0\ a2 \uparrow 2 \uparrow 0\ a0 \uparrow 0\ a0\ a2\ a2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0\}$.

The $RBCS\_post\_ranked\_bar(p_{10r})$ abbreviated as $RBCS$ contains the following items: $RBCS[a3] = 1$, $RBCS[a2] = 1$, $RBCS[a1] = 1$, $RBCS[a0] = 1$, $RBCS[\uparrow 3] = 6$, $RBCS[\uparrow 2] = 5$, $RBCS[\uparrow 1] = 4$, $RBCS[\uparrow 0] = 2$.

A trace of the run of the Algorithm 30 (PostfixReversedBackwardLTPM) is depicted in Table 5.7. Longer subtrees in place of wildcards S are denoted by $\uparrow S \rightarrow \leftarrow S$. △

The trace of the run of Algorithm 30 (PostfixReversedBackwardLTPM) from Example 5.5.4 is similar to the trace of the run of Algorithm 24 (BackwardLTPM) from

Example 5.2.11. The similarity is caused by the similarity in the notations used – prefix and postfix ranked bar notations. The length of the shift clearly depends on the position of the symbol S in the pattern in the same way as in case of the backward linearised tree pattern matching algorithm.

Algorithm properties stay the same again. The space complexity of the algorithm $\Theta(|\mathcal{A}|)$ is given by the size of the postfix reversed bad character shift table. The preprocessing time is $O(m + |\mathcal{A}|)$, where m is the pattern length and $|\mathcal{A}|$ is the alphabet size.

Similarly to the situation with prefix ranked bar notation, the expectation with postfix ranked bar notation is that again the reversed variant of the algorithm should be more efficient. The benefit of the reversed variant comes from the fact that non-bar symbols are compared sooner than bar symbols.

5.6 Backward linearised tree pattern matching on ranked notations

Bar symbols in the prefix (postfix) ranked bar notation are redundant. The information about tree structure is encoded in both the ranks and bar symbols. It is possible to apply principles of the backward linearised tree pattern matching introduced in Section 5.3 and Section 5.5 to design a similar algorithm that works with the prefix (postfix) notation. The content of this section is novel and not presented in any publication.

Considering the principles of construction prefix (postfix) notations of trees, it is practical to consider one direction of symbol comparison while checking particular position within the subject for occurrence only. The practical direction of symbol comparison is left to right for prefix notation and right to left for postfix notation, respectively. The prefix notation is not practical in case of the right to left symbol comparison because while skipping a subtree in place of the wildcard symbol, there can be more subtrees ending exactly at such a position. Similarly for postfix notation and symbol comparison from left to right. In other words, mapping from the position of the root of the subtree to the subtree's ending position in the prefix (postfix) ranked notation is a function but mapping from subtree's ending position to position of the root is not a function.

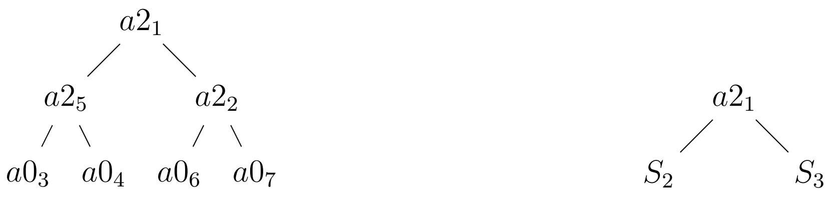
Example 5.6.1. Consider a ranked alphabet $\mathcal{A} = \{a2,\ a1,\ a0\}$. Consider an ordered, ranked, labelled, rooted, and directed tree t_{4r} in prefix notation $pref(t_{4r}) = a2\ a2\ a0\ a0\ a2\ a0\ a0$ over an alphabet $\mathcal{A}$.

Consider a tree pattern p_{10r} over an alphabet $\mathcal{A} \cup \{S\}$ in prefix notation $pref(p_{10r}) = a2\ S\ S$. Tree pattern p_{10r} is illustrated in Figure 5.6b.

Two possible sizes of subtrees that correspond to wildcard S and two occurrences of the pattern p_{10r} ending on single position are represented in Table 5.8. It can be seen that when comparing symbols from right to left, the tree pattern matching algorithm would have to try all possible subtree sizes in order not to miss any occurrence. △

Table 5.8: Representation of two possible occurrences of tree pattern p_{10r} ending on position 7 in tree t_{4r}.

subtree node id	1	2	3	4	5	6	7
$pref(p_{10r})$	$a2$	$a2$	$a0$	$a0$	$a2$	$a0$	$a0$
first occurrence	$a2$	$S\!\rightarrow$		$\leftarrow\! S$	$S\!\rightarrow$		$\leftarrow\! S$
second occurrence					$a2$	S	S



(a) Tree t_{4r} from Example 5.6.1. (b) Tree pattern p_{10r} from Example 5.6.1.

Figure 5.6: Tree t_{4r} (left) and tree pattern p_{10r} (right) from Example 5.6.1.

Hence it is possible to apply the principles from the backward linearised tree pattern matching variants introduced in Section 5.3 and Section 5.5 to work on prefix and postfix notation, respectively. However, not the algorithms from Section 5.2 and Section 5.5. Those algorithms would require more complicated occurrence checks on the prefix (postfix) notation in practice, as shown in Example 5.6.1.

5.6.1 Prefix notation

The backward linearised tree pattern matching for prefix notation can be defined similarly to the reversed variant of backward linearised tree pattern matching for prefix ranked bar notation.

Definition 5.6.2. Let $pattern[1..m]$ be a prefix notation of a tree pattern p over an alphabet $\mathcal{A}$. The *bad character shift table for prefix notation $BCS\_pref(p)$* for the backward linearised tree pattern matching on the prefix notation is defined for each $a \in \mathcal{A}$.

$$BCS\_pref(p)[a] = \min \left(\begin{array}{l} \{m\} \cup \{j-1 : pattern[j] = a \text{ and } m \geq j > 1\} \cup \\ \{j-1 : pattern[j] = S \text{ and } m \geq j > 1\} \end{array} \right)$$

Definition 5.6.2 is similar to Definition 5.3.1. There are no bar symbols in the prefix notation, therefore, the definition of shift concerning the wildcard symbol S is simpler. However, all symbols can be the root of a subtree in place of the wildcard symbol S. Meaning, the maximal length of the shift is directly limited by the first occurrence of the wildcard symbol S in the prefix notation of the tree.

```
Name: ConstructBCS_pref
Input: The tree pattern in the prefix notation $pref(pattern)$ of size $m$ over an
       alphabet $\mathcal{A}$ of the subject tree
Result: The bad character shift table $BCS\_pref(pattern)$
begin
    $s := m$;
    for $i := m$ downto 1 do
        if $pref(pattern)[i] = S$ then $s := i - 1$;
    end
    if $s = 0$ then $s := 1$;
    foreach $x \in \mathcal{A}$ do
        if $BCS\_pref(pattern)[x] > s$ then
            $BCS\_pref(pattern)[x] := s$;
        end
    end
    for $i := m$ downto 2 do
        if $pref(pattern)[i] \neq S$ and $BCS\_pref(pattern)[pref(pattern)[i]] > (i - 1)$
          then
            $BCS\_pref(pattern)[pref(pattern)[i]] := i - 1$;
        end
    end
    return $BCS\_pref(pattern)$;
end
```

Algorithm 31: Construction of bad character shift table for prefix notation.

Algorithm 31 (ConstructBCS_pref) of construction of $BCS\_pref(pattern)$ for the backward linearised tree pattern matching on the prefix notation first searches for the firstly occurrence of wildcard symbol S. Secondly, it limits the shift by the position of first wildcard symbol S (and by the length of the pattern m), and finally it limits the shift for individual symbols by their minimal distance from the beginning of the pattern if they are used in the pattern.

Example 5.6.3. Consider a tree pattern p_{8r}, depicted in Figure 5.1, in the prefix notation $pref(p_{8r}) = a2\ a1\ S\ a1\ a0$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S\}$. The bad character shift table for prefix notation $BCS\_pref(p_{8r})$ abbreviated as BCS constructed by Algorithm 31 (ConstructBCS_pref) contains the following items. △

$$\begin{array}{ll} BCS[a3] = \min(\{5\} \cup \emptyset \cup \{2\}) = 2 & BCS[a2] = \min(\{5\} \cup \emptyset \cup \{2\}) = 2 \\ BCS[a1] = \min(\{5\} \cup \{1,3\} \cup \{2\}) = 1 & BCS[a0] = \min(\{5\} \cup \{4\} \cup \{2\}) = 2 \end{array}$$

The tree pattern p_{8r} used in Example 5.6.3 is the same as in Example 5.2.5. The length of the shift is limited by the occurrence of the first wildcard symbol S. Any symbol can

Table 5.9: Subtree jump table for prefix notation $SJT\_pref(t_{4r})$ of tree t_{4r}.

node id	1	2	3	4	5	6	7
$pref(t_{4r})$	$a2$	$a2$	$a0$	$a0$	$a2$	$a0$	$a0$
$SJT\_pref(t_{4r})$	8	5	4	5	8	7	8

be a root of a subtree matched to wildcard symbol S, hence the lengths of shifts can't be extended further.

A subtree jump table for prefix notation (SJT_pref) is used by the matching algorithm. Similarly as with previously defined subtree jump tables, the SJT_pref stores a position where to continue matching when a subtree is skipped. The definition of the subtree jump table for prefix notation (SJT_pref) is in Definition 4.5.1.

Example 5.6.4. Consider a tree t_{4r} in the prefix notation $pref\ (t_{4r}) = a2\ a2\ a0\ a0\ a2\ a0\ a0$ over alphabet $\mathcal{A} = \{a3, a2, a1, a0\}$. Table 5.9 shows the $SJT\_pref(t_{4r})$. △

```
Name: PrefixRankedBackwardLTPM
Input: The subject tree in the prefix notation $pref(subject)$ of size $n$
Input: The subtree jump table for prefix notation $SJT\_pref(subject)$
Input: The tree pattern in the prefix notation $pref(pattern)$ of size $m$
Input: The bad character shift table $BCS\_pref(pattern)$
Result: Locations of occurrences of tree pattern $pattern$ in subject tree $subject$
begin
    $i := n - m + 1$;
    while $i >= 1$ do
        $j = 1$;
        $position = i$;
        while $j <= m$ and $position <= n$ do
            if $pref(subject)[position] = pref(pattern)[j]$ then
                $position := position + 1$;
            else if $pref(pattern)[j] = S$ then
                $position := SJT\_pref(subject)[position]$;
            else break;
            $j := j + 1$;
        end
        if $j = m + 1$ then yield $i$;
        $i := i - BCS\_pref(pattern)[pref(subject)[i]]$;
    end
end
```

Algorithm 32: Backward linearised tree pattern matching algorithm on prefix notation.

Table 5.10: A trace of the run of Algorithm 32 (PrefixRankedBackwardLTPM) for subject tree t_{4r} and tree pattern p_{10r}.

1	2	3	4	5	6	7	
$a2$	$a2$	$a0$	$a0$	$a2$	$a0$	$a0$	$pref(t_{4r})$
8	5	4	5	8	7	8	$subtree\_jumps(t_{4r})$
				$a2$	S	S	match, $shift = 1$
			$a2$				$a0 \neq a2$, $shift = 1$
		$a2$					$a0 \neq a2$, $shift = 1$
	$a2$	S	S				match, $shift = 1$
$a2$	$S{\rightarrow}$		${\leftarrow}S$	$S{\rightarrow}$		${\leftarrow}S$	match, $shift = 1$

Algorithm 32 (PrefixRankedBackwardLTPM) is similar to Algorithm 26 (Reversed-BackwardLTPM). The subtree jump table and bad character shift table are exchanged for SJT_pref and BCS_pref, respectively, for the algorithm to work, and since the length of the representation of wildcard symbol S in prefix notation is 1, the extra decrementation of variable j is removed from handling subtree skips.

Theorem 5.6.5. Given a tree pattern p in the prefix notation and shift table BCS_pref (p) constructed for the tree pattern p by Algorithm 31 (ConstructBCS_pref), Algorithm 32 (PrefixRankedBackwardLTPM) correctly computes the locations of all occurrences of tree pattern p in subject tree t.

Proof. The proof is similar to the proof of Theorem 5.2.10 and it is presented in an appendix as proof of Theorem A.1.4. □

Example 5.6.6. Consider a tree pattern p_{10r} in the prefix notation $pref(p_{10r}) = a2\ S\ S$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S\}$ and a tree t_{4r} in the prefix notation $pref\ (t_{4r}) = a2\ a2\ a0\ a0\ a2\ a0\ a0$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0\}$.

The $BCS\_pref(p_{10r})$ abbreviated as BCS contains the following items: $BCS[a3] = 1$, $BCS[a2] = 1$, $BCS[a1] = 1$, $BCS[a0] = 1$.

A trace of the run of Algorithm 32 (PrefixRankedBackwardLTPM) is depicted in Table 5.10. Longer subtrees in place of wildcards S are denoted by $S{\rightarrow}\ {\leftarrow}S$. △

The run of Algorithm 32 (PrefixRankedBackwardLTPM) from Example 5.6.6 is similar to the run of Algorithm 26 (ReversedBackwardLTPM) from Example 5.3.5. Shift direction and symbol comparison direction are the same as in Algorithm 26 (Reversed-BackwardLTPM). Bar symbols are omitted in the prefix notation of a tree. Hence the prefix notation is shorter than prefix ranked bar notation, which is beneficial for the algorithm as the amount of data that needs to be processed is smaller. Shift lengths, on the other hand, are shorter as well since they are limited exactly by the position of the first wildcard symbol. These two properties work against each other. Moreover they are expected to cancel each other out.

Properties of the algorithm stay the same. The space complexity of the algorithm $\Theta(|\mathcal{A}|)$ is given by the size of the bad character shift table for prefix notation. The preprocessing time is $O(m + |\mathcal{A}|)$, where m is the pattern length and $|\mathcal{A}|$ is the alphabet size.

5.6.2 Postfix notation

The backward linearised tree pattern matching for postfix notation can be defined similarly, however, taking the reversed variant of the backward linearised tree pattern matching for postfix ranked bar notation as the base variant of the approach.

Definition 5.6.7. Let $pattern[1..m]$ be a *post* notation of a tree pattern p over an alphabet $\mathcal{A}$. The *bad character shift table for postfix notation* $BCS\_post(p)$ for backward linearised tree pattern matching on postfix notation is defined for each $a \in \mathcal{A}$.

$$BCS\_post(p)[a] = \min \begin{pmatrix} \{m\} \cup \{j : pattern[m-j] = a \text{ and } m > j > 0\} \cup \\ \{j : pattern[m-j] = S \text{ and } m > j > 0\} \end{pmatrix}$$

Definition 5.6.7 is similar to Definition 5.5.1. Similarly as Definition 5.6.2 was modified, any symbol can be the root of a subtree in place of the wildcard symbol, therefore the maximal length of the shift is similarly limited directly by the last occurrence of the wildcard symbol in the postfix notation of the tree. There are no bar symbols in the postfix notation again, hence the definition of shift concerning the wildcard symbol S is simpler, resulting, however, in shorter shifts.

Algorithm 33 (ConstructBCS_post) of construction for $BCS\_post(pattern)$ for the backward linearised tree pattern matching on the postfix notation firstly searches for the last occurrence of wildcard symbol S. Secondly it limits the shift by the position of last wildcard symbol S (and by the length of the pattern m), and finally it limits the shift for individual symbols by their minimal distance from the end of the pattern if they are used in the pattern.

Example 5.6.8. Consider a tree pattern p_{8r}, depicted in Figure 5.1, in the postfix notation $post(p_{8r}) = S\ a1\ a0\ a1\ a2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S\}$. The bad character shift table for postfix notation $BCS\_post(p_{8r})$ abbreviated as BCS constructed by Algorithm 33 (ConstructBCS_post) contains the following items. △

$$\begin{array}{ll} BCS[a3] = \min(\{5\} \cup \emptyset \cup \{4\}) = 4 & BCS[a2] = \min(\{5\} \cup \emptyset \cup \{4\}) = 4 \\ BCS[a1] = \min(\{5\} \cup \{1,3\} \cup \{4\}) = 1 & BCS[a0] = \min(\{5\} \cup \{2\} \cup \{4\}) = 2 \end{array}$$

The tree pattern p_{8r} used in Example 5.6.8 is the same as in Example 5.2.5. The length of the shift is limited by the occurrence of the last wildcard symbol S and any symbol can be a root of a subtree matched to the wildcard symbol S, hence the lengths of shifts can't be extended further.

```
Name: ConstructBCS_post
Input: The tree pattern in the postfix notation $post(pattern)$ of size $m$ over an
       alphabet $\mathcal{A}$ of the subject tree
Result: The bad character shift table $BCS\_post(pattern)$
begin
    $s := m$;
    for $i := 1$ to $m$ do
        if $post(pattern)[i] = S$ then $s := m - i$;
    end
    if $s = 0$ then $s := 1$;
    foreach $x \in \mathcal{A}$ do
        if $BCS\_post(pattern)[x] > s$ then
            $BCS\_post(pattern)[x] := s$;
        end
    end
    for $i := 1$ to $m - 1$ do
        if $post(pattern)[i] \neq S$ and $BCS\_post(pattern)[post(pattern)[i]] > (m - i)$
          then
            $BCS\_post(pattern)[post(pattern)[i]] := m - i$;
        end
    end
    return $BCS\_post(pattern)$;
end
```

Algorithm 33: Construction of bad character shift table for postfix notation.

A subtree jump table for postfix notation (SJT_post) is used by the matching algorithm. Similarly as with previously defined subtree jump tables, the SJT_post stores a position where to continue matching when a subtree is skipped.

Definition 5.6.9. Let t and $post(t)$ of length n be a tree and its postfix notation, respectively. A *subtree jump table for postfix notation* denoted as $SJT\_post(t)$ is defined as a mapping from set of integers $\{1..n\}$ into a set of integers $\{0..n-1\}$. If $post(t)$ $[i..j]$ is the *post* notation of a subtree of tree t, then $SJT\_post(t)[j] = i - 1$, $1 \leq i < j \leq n$.

The subtree jump table for postfix notation is similar to the subtree jump table defined for the prefix notation in Definition 4.5.1. The subtree jump table for postfix notation allows to jump only from a position of a root of a subtree t to a closest position after the beginning of the linear representation of the subtree t.

Lemma 5.6.10. Given tree t in postfix notation $post(t)$ and initial value of $rootIndex$ equal to n, Algorithm 34 (ConstructSJT_post) constructs subtree jump table for postfix notation $SJT\_post(t)$.

```
Name: ConstructSJT_post
Input: Tree $t$ in postfix notation $post(t)$ of size $n$
Input: Index of current node $rootIndex$ initialised to $n$
Input/Output: Subtree jump table for postfix notation $SJT\_post(t)$ initialised
              to empty array of length $n$
Result: Index $exitIndex$
1 begin
2     $index := rootIndex - 1$;
3     for $i = 1$ to $Arity(post(t)[rootIndex])$ do
4         $index := ConstructSJT\_post(post(t), index, SJT\_post(t))$;
5     end
6     $SJT\_post(t)[rootIndex] = index$;
7     return $index$;
8 end
```

Algorithm 34: Construction of subtree jump table for postfix notation.

Table 5.11: Subtree jump table for postfix notation $SJT\_post(t_{4r})$ of tree t_{4r}.

node id	1	2	3	4	5	6	7
$post(t_{4r})$	$a0$	$a0$	$a2$	$a0$	$a0$	$a2$	$a2$
$SJT\_post(t_{4r})$	0	1	0	3	4	3	0

Example 5.6.11. Consider a tree t_{4r} in the postfix notation $post\ (t_{4r}) = a0\ a0\ a2\ a0\ a0\ a2\ a2$ over alphabet $\mathcal{A} = \{a3, a2, a1, a0\}$. Table 5.11 shows the $SJT\_post(t_{4r})$. △

Algorithm 35 (PostfixRankedBackwardLTPM) is similar to Algorithm 30 (PostfixReversedBackwardLTPM). The subtree jump table and the bad character shift table are exchanged for SJT_post and BCS_post, respectively. Since the length of the representation of wildcard symbol S in postfix notation is 1, the extra decrementation of variable j is removed from handling subtree skips, again.

Theorem 5.6.12. Given a tree pattern p in the postfix notation and the shift table $BCS\_post(p)$ constructed for the tree pattern p by Algorithm 33 (ConstructBCS_post), Algorithm 35 (PostfixRankedBackwardLTPM) correctly computes the locations of all occurrences of tree pattern p in subject tree t.

Proof. The proof is similar to the proof of Theorem 5.2.10 and it is presented in an appendix as proof of Theorem A.1.5. □

Example 5.6.13. Consider a tree pattern p_{10r} in the postfix notation $post(p_{10r}) = S\ S\ a2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0, S\}$ and a tree t_{4r} in the postfix notation $post\ (t_{4r}) = a0\ a0\ a2\ a0\ a0\ a2\ a2$ over an alphabet $\mathcal{A} = \{a3, a2, a1, a0\}$.

```
Name: PostfixRankedBackwardLTPM
Input: The subject tree in the postfix notation $post(subject)$ of size $n$
Input: The subtree jump table for postfix notation $SJT\_post(subject)$
Input: The tree pattern in the postfix notation $post(pattern)$ of size $m$
Input: The bad character shift table $BCS\_post(pattern)$
Result: Locations of occurrences of the pattern $pattern$ in the tree $subject$
begin
    $i := 0$;
    while $i <= (n - m)$ do
        $j := m$;
        $position := i + j$;
        while $j > 0$ and $position > 0$ do
            if $post(subject)[position] = post(pattern)[j]$ then
                $position := position - 1$;
            else if $post(pattern)[j] = S$ then
                $position := SJT\_post(subject)[position]$;
            else break;
            $j := j - 1$;
        end
        if $j = 0$ then yield $position + 1$;
        $i := i + BCS\_post(pattern)[post(subject)[i + m]]$;
    end
end
```

Algorithm 35: Backward linearised tree pattern matching algorithm on postfix notation.

The $BCS\_post(p_{10r})$ abbreviated as BCS contains the following items: $BCS[a3] = 1$, $BCS[a2] = 1$, $BCS[a1] = 1$, $BCS[a0] = 1$.

A trace of the run of Algorithm 35 (PostfixRankedBackwardLTPM) is depicted in Table 5.12. Longer subtrees in place of wildcards S are denoted by $S\!\rightarrow\ \leftarrow\!S$. $\triangle$

The run of Algorithm 35 (PostfixRankedBackwardLTPM) from Example 5.6.13 is similar to the run of Algorithm 30 (PostfixReversedBackwardLTPM) from Example 5.5.4. Shift direction and symbol comparison direction stay the same as in Algorithm 30 (PostfixReversedBackwardLTPM). Bar symbols are omitted in the postfix notation. Hence the postfix notation is shorter than the postfix ranked bar notation, which is beneficial for the algorithm as the amount of data that needs to be processed is smaller. As in the case of prefix notation, the shift lengths, on the other hand, are shorter as well since they are still limited exactly by the position of the last wildcard symbol. These two properties again work against each other and they are still expected to cancel each other out.

The algorithm properties stay the same. The space complexity of the algorithm $\Theta(|\mathcal{A}|)$ is given by the size of the bad character shift table for postfix notation. The preprocessing time is $O(m + |\mathcal{A}|)$, where m is the pattern length and $|\mathcal{A}|$ is the alphabet size.

Table 5.12: A trace of the run of Algorithm 35 (PostfixRankedBackwardLTPM) for subject tree t_{4r} and tree pattern p_{10r}.

1	2	3	4	5	6	7	
$a0$	$a0$	$a2$	$a0$	$a0$	$a2$	$a2$	$post(t_{4r})$
0	1	0	3	4	3	0	$subtree\_jumps(t_{4r})$
S	S	$a2$					match, $shift = 1$
			$a2$				$a0 \neq a2$, $shift = 1$
		$a2$		$a2$			$a0 \neq a2$, $shift = 1$
			S	S	$a2$		match, $shift = 1$
$S\!\rightarrow$		$\leftarrow\!S$	$S\!\rightarrow$		$\leftarrow\!S$	$a2$	match, $shift = 1$

5.7 Nonlinear backward linearised tree pattern matching

This section contains an original extension of the backward linearised tree pattern matching algorithm from the previous sections to handle nonlinear tree templates as queries as well. The content is novel and not presented in any publication.

As in the case of methods for indexing trees, the nonlinear tree templates are considered as input for a pattern matching algorithm. A modification of presented backward linearised tree pattern matching algorithms adapted for locating occurrences of nonlinear tree templates is presented in this chapter as *Nonlinear backward linearised tree pattern matching algorithm*. The algorithm requires a modification of the bad character shift table construction algorithm, which is also presented.

The basic idea of the nonlinear backward linearised tree pattern matching algorithm is the same as the idea of backward linearised tree pattern matching algorithm. However, in addition to the wildcard S, there are nonlinear variables $X, Y, \ldots$ present in the tree pattern. The same nonlinear variable is, in an occurrence, located in place of the same subtrees of the subject. The backward linearised tree pattern matching algorithm modified for locating nonlinear tree patterns additionally requires a subtree repeats table. This table can be obtained by algorithm presented in [11] extended with Algorithm 8 (PostfixToPrefixRepeats), Algorithm 9 (PostfixToPrefixBarRepeats) presented in Section 3.4, or any other subtree repeats table transformation algorithm, given the used linear notation of a tree.

The presented nonlinear backward linearised tree pattern algorithm modification uses the prefix ranked bar notation, but other previously presented modifications of the backward linearised tree pattern matching algorithm can be modified in the same manner as is presented in this chapter. Hence the selection of linear tree notation is not important and the nonlinear backward linearised tree pattern matching algorithm is working with all of them.

Definition 5.7.1. Let $pattern[1..m]$ be a $pref\_ranked\_bar$ notation of tree pattern p over alphabet $\mathcal{A}$. The *nonlinear bad character shift table* $NonlinearBCS\_pref\_ranked\_bar(p)$

for the nonlinear backward linearised tree pattern matching is defined for each $a \in \mathcal{A}$. Let $T \in \{S\} \cup \mathcal{X}$.

$$NonlinearBCS\_pref\_ranked\_bar(p)[a] = \min \begin{pmatrix} \{m\} \cup \{j : pattern[m-j] = a \text{ and } m > j > 0\} \cup \\ \{j + Arity(a) * 2 : pattern[m-j] = T \text{ and } m > j > 0 \text{ and } a \notin \mathcal{A}_{\uparrow}\} \cup \\ \{j - 1 : pattern[m-j] = T \text{ and } m > j > 0 \text{ and } a \in \mathcal{A}_{\uparrow} \text{ and } pattern \neq T \uparrow T\} \cup \\ \{1 : a \in \mathcal{A}_{\uparrow} \text{ and } pattern = T \uparrow T\} \end{pmatrix}$$

```
Name: ConstructNonlinearBCS_pref_ranked_bar
Input: Nonlinear tree pattern in the prefix ranked bar notation
       $pref\_ranked\_bar(pattern)$ of size $m$ over alphabet $\mathcal{A}$ of the subject tree
Result: The bad character shift table $NonlinearBCS\_pref\_ranked\_bar(pattern)$
begin
    $s := m$;
    for $i := 1$ to $m$ do
        if $pref\_ranked\_bar(pattern)[i] \in (\{S\} \cup \mathcal{X})$ then $s = m - i$;
    end
    foreach $x \in \mathcal{A}$ do $NonlinearBCS\_pref\_ranked\_bar(pattern)[x] = m$;
    foreach $x \in \mathcal{A}$ do
        if $x \notin \mathcal{A}_{\uparrow}$ then $shift := s + Arity(x) * 2$;
        else if $s >= 2$ then $shift := s - 1$;
        else $shift := s$;
        if $NonlinearBCS\_pref\_ranked\_bar(pattern)[x] > shift$ then
            $NonlinearBCS\_pref\_ranked\_bar(pattern)[x] := shift$;
        end
    end
    for $i := 1$ to $m - 1$ do
        if $pref\_ranked\_bar(pattern)[i] \notin (\{S, \uparrow S\} \cup \mathcal{X})$ and
         $NonlinearBCS\_pref\_ranked\_bar(pattern)[pref\_ranked\_bar(pattern)[i]] >$
         $(m - i)$ then
            $NonlinearBCS\_pref\_ranked\_bar(pattern)[pref\_ranked\_bar(pattern)[i]] :=$
              $m - i$;
        end
    end
    return $NonlinearBCS\_pref\_ranked\_bar(pattern)$;
end
```

Algorithm 36: Construction of the nonlinear bad character shift table.

The nonlinear variables, in essence, behave similarly as the wildcard. They can be treated the same way in the construction of bad character shift table. There can be nothing known about the subtree matched to the nonlinear variable as well except for its minimal size. Considering this modification of the construction of bad character shift table for nonlinear tree templates is straightforward. Wherever a wildcard is used in a condition a set of nonlinear variables is also used.

The modification of the backward linearised tree pattern matching for nonlinear tree templates is utilising the same idea of nonlinear variables being similar to the wildcard. Nonlinear variables are matched to a subtree of the subject which shall be skipped using the subtree jump table. However, due to the nature of nonlinear variables where all occurrences of the same nonlinear variable in one particular match must correspond to the same subtree, an additional check must be done in the algorithm. In each match attempt, a map of the current setting for all nonlinear variable is maintained. This setting can only be initialised when the nonlinear variables are processed first; later it is only compared for consistency.

Note that the subtree repeats table SRT parameter of the Algorithm 37 (Nonlinear-BackwardLTPM) is correctly its variant for the prefix bar notation. The prefix bar notation and the prefix ranked bar notation share structure. Hence the prefix bar notation variant of the subtree repeats table is correct.

Theorem 5.7.2. Given nonlinear tree pattern p in the prefix ranked bar notation and bad character shift table $BCS\_pref\_ranked\_bar(p)$ constructed for the tree pattern p by Algorithm 36 (ConstructNonlinearBCS_pref_ranked_bar), Algorithm 37 (NonlinearBackwardLTPM) correctly computes the locations of all occurrences of nonlinear tree pattern p in subject tree t.

Proof. The nonlinear backward linearised tree pattern matching algorithm for prefix ranked bar notation is an extension of the backward linearised tree pattern matching algorithm. The shifting of the pattern based on $BCS\_pref\_ranked\_bar(p)$ is unchanged. The checks of individual positions inside the subject tree for occurrences are additionally extended with consistency checks of nonlinear variables $X, Y, \ldots$ using a mapping *variables* from a nonlinear variable to a subtree repeat identifier. The consistency check of nonlinear variables is performed when a subtree corresponding to nonlinear variable $X{\uparrow}X$, $Y{\uparrow}Y$, ... is skipped. Subtree repeats table $SRT\_pref\_bar(t)$ stores the same identifiers at start positions of the same subtrees. At each occurrence check, these identifiers are used at each skip of a subtree corresponding to any nonlinear variable and the mapping *variables* can only be set once for each nonlinear variable. Later the mapping *variables* can only be read to test the consistency of skipped subtrees by individual nonlinear variables. Hence the consistency check can detect a mismatch between the tree pattern and a subtree of the subject tree at any position resulting even from an inconsistency of subtrees skipped by nonlinear variables. $\square$

Example 5.7.3. Consider a nonlinear tree pattern p_{12r} in the prefix ranked bar notation $pref\_ranked\_bar(p_{12r}) = a2\ X\ {\uparrow}X\ X\ {\uparrow}X\ {\uparrow}2$ over an alphabet $\mathcal{A} = \{a3,\ a2,\ a1,\ a0,\ S,\ X,\ {\uparrow}3,\ {\uparrow}2,\ {\uparrow}1,\ {\uparrow}0,\ {\uparrow}S,\ {\uparrow}X\}$ depicted in Figure 5.7 and a tree t_{5r} in the prefix ranked bar notation $pref\_ranked\_bar\ (t_{5r}) = a2\ a2\ b0\ {\uparrow}0\ a0\ {\uparrow}0\ {\uparrow}2\ a2\ a0\ {\uparrow}0\ a0\ {\uparrow}0\ {\uparrow}2\ {\uparrow}2$ over an alphabet $\mathcal{A} = \{a3,\ a2,\ a1,\ a0,\ b0,\ {\uparrow}3,\ {\uparrow}2,\ {\uparrow}1,\ {\uparrow}0\}$.

```
Name: NonlinearBackwardLTPM
Input: The subject tree in $pref\_ranked\_bar(subject)$ notation of size $n$
Input: The tree pattern in $pref\_ranked\_bar(pattern)$ notation of size $m$
Input: The subtree jump table $SJT\_bar(subject)$
Input: The bad character shift table $NonlinearBCS\_pref\_ranked\_bar(pattern)$
Input: The $SRT\_pref\_bar(subject)$ representation of repetitions in the subject
Result: Locations of occurrences of the pattern pattern in the tree subject
begin
    $i := 0$;
    while $i <= (n - m)$ do
        $j := m$;
        $position := i + j$;
        $variables$ := empty mapping from nonlinear variable to subtree repeat identifier;
        while $j > 0$ and $position > 0$ do
            if $pref\_ranked\_bar(subject)[position] = pref\_ranked\_bar(pattern)[j]$ then
                $position := position - 1$;
            else if $pref\_ranked\_bar(pattern)[j] = \uparrow S$ and $pref\_ranked\_bar(subject)[position] \in \mathcal{A}_\uparrow$ then
                $position := SJT\_bar(subject)[position]$;
                $j = j - 1$; {Subtree skip} if $pref\_ranked\_bar(pattern)[j] \in \mathcal{X}$ then
                    {Check nonlinear variable}
                    if is not set $variables[pref\_ranked\_bar(pattern)[j]]$ then
                        $variables[pref\_ranked\_bar(pattern)[j]] = SRT\_pref\_bar(subject)[position + 1]$;
                    else if $variables[pref\_ranked\_bar(pattern)[position + 1]] \neq SRT\_pref\_bar(subject)[position + 1]$ then
                        break;
            else break;
            $j := j - 1$;
        end
        if $j = 0$ then yield $position + 1$;
        $i := i + NonlinearBCS\_pref\_ranked\_bar(pattern)[pref\_ranked\_bar(subject)[i + m]]$;
    end
end
```
Algorithm 37: Nonlinear backward linearised tree pattern matching algorithm.

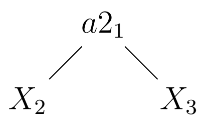


Figure 5.7: Nonlinear tree pattern p_{12r} from Example 5.7.3.

Table 5.13: A trace of the run of Algorithm 37 (NonlinearBackwardLTPM) for subject tree t_{5r} and tree pattern p_{12r}.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	
$a2$	$a2$	$b0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$	$pref\_ranked\_bar(t_{5r})$
14	6	2	2	2	2	6	6	2	2	2	2	6	14	$subtree\_jumps(t_{5r})$
					$\uparrow 2$									$\uparrow 0 \neq \uparrow 2$, $shift = 1$
		$X\rightarrow$	$\leftarrow\uparrow X$	$X\rightarrow$	$\leftarrow X\uparrow$	$\uparrow 2$								inconsistent, $shift = 1$
							$\uparrow 2$							$a2 \neq \uparrow 0$, $shift = 5$
							$a2$	$X\rightarrow$	$\leftarrow\uparrow X$	$X\rightarrow$	$\leftarrow\uparrow X$	$\uparrow 2$		match, $shift = 1$
	$X\rightarrow$					$\leftarrow\uparrow X$	$X\rightarrow$					$\leftarrow\uparrow X$	$\uparrow 2$	inconsistent, $shift = 1$

The $NonlinearBCS\_pref\_ranked\_bar(p_{12r})$ abbreviated as BCS contains the following items: $BCS[a3] = 6$, $BCS[a2] = 5$, $BCS[a1] = 4$, $BCS[a0] = 2$, $BCS[b0] = 2$, $BCS[\uparrow 3] = 1$, $BCS[\uparrow 2] = 1$, $BCS[\uparrow 1] = 1$, $BCS[\uparrow 0] = 1$.

A trace of the run of Algorithm 37 (NonlinearBackwardLTPM) is depicted in Table 5.13. Longer subtrees in place of nonlinear variables X are denoted by $X\rightarrow \leftarrow\uparrow X$. $\triangle$

The trace of the run of Algorithm 37 (NonlinearBackwardLTPM) presented in Example 5.7.3 shows how the algorithm locates all occurrences of the nonlinear tree pattern p_{12r} inside the subject tree t_{5r}. The trace of the run of Algorithm 37 (NonlinearBackwardLTPM) is almost identical to the trace of the run of Algorithm 24 (BackwardLTPM) because both, the nonlinear tree pattern p_{12r} and the subject tree t_{5r} used as an input of Algorithm 37 (NonlinearBackwardLTPM) are structurally similar when compared to tree pattern p_{10r} and subject tree t_{4r} used as an input of Algorithm 24 (BackwardLTPM). The difference of the two traces of runs is related to the exchange of the wildcard symbol S of the pattern for nonlinear variable X and one of the leaves $a0$ of the subject tree for a leaf $b0$.

A match attempt can in the case of Algorithm 37 (NonlinearBackwardLTPM) report a mismatch in a new situation (referred to as inconsistent in the trace of the run table). The situation is related to a nonlinear variable, more precisely when the subtrees they are substituting would not be the same.

The nonlinear backward linearised tree pattern matching algorithm keeps properties of the backward linearised tree pattern matching algorithm, most importantly the running time complexity. The algorithm can still perform a sublinear number of comparisons given it is provided with subtree repeats table SRT_pref_bar.

5.8 Some empirical results

We have implemented our tree pattern matching algorithms by extending the existing Forest FIRE toolkit and accompanying FIRE Wood graphical user interface [12, 59]. This toolkit already implemented many tree pattern matching algorithms and constructions of automata used in them, but no algorithms based on linearisations of both pattern tree(s) and subject tree(s). Constructions included in Forest FIRE include ones described in [2, 10, 13, 36] and others. We compared our algorithms' performances to some of the best-performing ones in Forest FIRE, according to the results in [13]. We compared the running times of the search phase of the following algorithms: 1) the new backward linearised tree pattern matching algorithms based on various linearisations of pattern and subject tree; 2) an algorithm based on the use of a *deterministic frontier-to-root (bottom-up) tree automaton* (DFRTA) constructed for the pattern; and 3) an algorithm based on the use of a *Aho-Corasick automaton* constructed for the pattern's stringpath set.

The comparison was made again using pattern sets previously used for benchmarking Forest FIRE as mentioned in Section 4.7.

Since our pattern matching algorithms are single-pattern ones, we ran each of the algorithms for each pattern individually, and sequentially ran it over each subject tree from two sets of subject trees: a set of 150 trees of approximately 500 nodes each and a set of 500 trees of approximately 150 nodes each. Both of these sets had previously been used for benchmarking Forest FIRE. Benchmarking was conducted on a 2 GHz Intel Core i7 with 16 GB of RAM running OpenSUSE GNU/Linux version 13.1 using Java SE 7.

Another run of the testing of our algorithms based on the tree linearisation was conducted using pattern sets with the subtree wildcard changed to a nonlinear variable, hence on nonlinear pattern sets. Algorithms originally implemented in Forest FIRE are unable to find occurrences of nonlinear tree patterns directly, and therefore they were not executed in the second phase of testing.

Linearised versions of the (nonlinear) tree patterns and subject trees were constructed from the in-memory tree representations, using additional memory. However, this representation is linear in the size of the tree representations, while the shift tables used will typically be much smaller than the automata used in the other algorithms. Because of this and because search time was our primary concern, we do not consider memory use. Figures 5.8 and 5.9 show the search times of patterns from the tree pattern sets as boxplots, showing that on average, our new algorithms considerably outperform existing ones for the single-pattern case (note the logarithmic scale).

The second run of testing of backward linearised tree pattern matching algorithms is shown in Figures 5.10 and 5.11. Figures show search times of nonlinear tree patterns from nonlinear pattern sets. The overall behaviour of algorithms modified for nonlinear tree patterns is similar to the original backward linearised tree pattern matching algorithms. The overall slowdown, i.e. a ratio of the running time of the nonlinear and original algorithm, introduced by nonlinear variables handling is plotted in Figures 5.12 and 5.13. The average slowdown is about 10 %–20 % (ratio 1.1–1.2), however sometimes the experimental evaluation of the nonlinear variant of the backward linearised tree pattern matching algorithms shows the ratio values smaller than 1, actually meaning a speedup. Such a speedup is caused by the detection of a mismatch in a match attempt caused by inconsistent setting of nonlinear variables.

Note that the computation of subtree repeats is not included in the search time.

5.9 Conclusion of the tree pattern matching

We have presented general approaches of linearised backward linearised tree pattern matching designed for the prefix and postfix ranked bar notation. The algorithms may perform a sublinear number of comparisons of symbols (labels) with respect to the size of the subject tree and perform well in practice. The algorithms can also work in reverse where the direction of shifting and symbol comparison are swapped.

Reversed and nonreversed approaches are symmetric, however, the reversed variants perform better for the given dataset. There are two possible causes. First, the patterns might be better suited for reversed backward linearised tree pattern matching because of the locations of variables and overall longer shifts. Second, the reversed variant starts every match attempt of a pattern within a subject on symbols rather than on bars. All symbols of given arity have the same corresponding bar of the same arity, therefore the number of different bar symbols is smaller or equal to the number of non-bar symbols. The reversed backward linearised pattern matching algorithm will detect mismatch sooner then non-reversed variants of backward linearised tree pattern matching algorithm.

We also adapted the backward linearised tree pattern matching algorithms to prefix and postfix notations. These variants only allow one of the shift and symbol comparison directions in practice and benefit from the shorter representation. On the other hand, the shifts are shorter. As testing shows both these properties cancel each other and the overall performance of algorithms working on prefix or postfix ranked bar notations and on prefix or postfix notations are comparable.

All of these backward linearised tree pattern matching algorithms can also be modified to locate occurrences of nonlinear tree patterns. Properties of the pattern matching algorithms modified for nonlinear tree patterns do not change. They can still perform a sublinear number of comparisons of symbols. According to the measurements, the additional handling of nonlinear variables adds on average 10 %–20 % of the running time. A sooner detection of a mismatch and in overall fewer occurrences can also make the nonlinear variant of backward linearised tree pattern matching algorithm run faster.

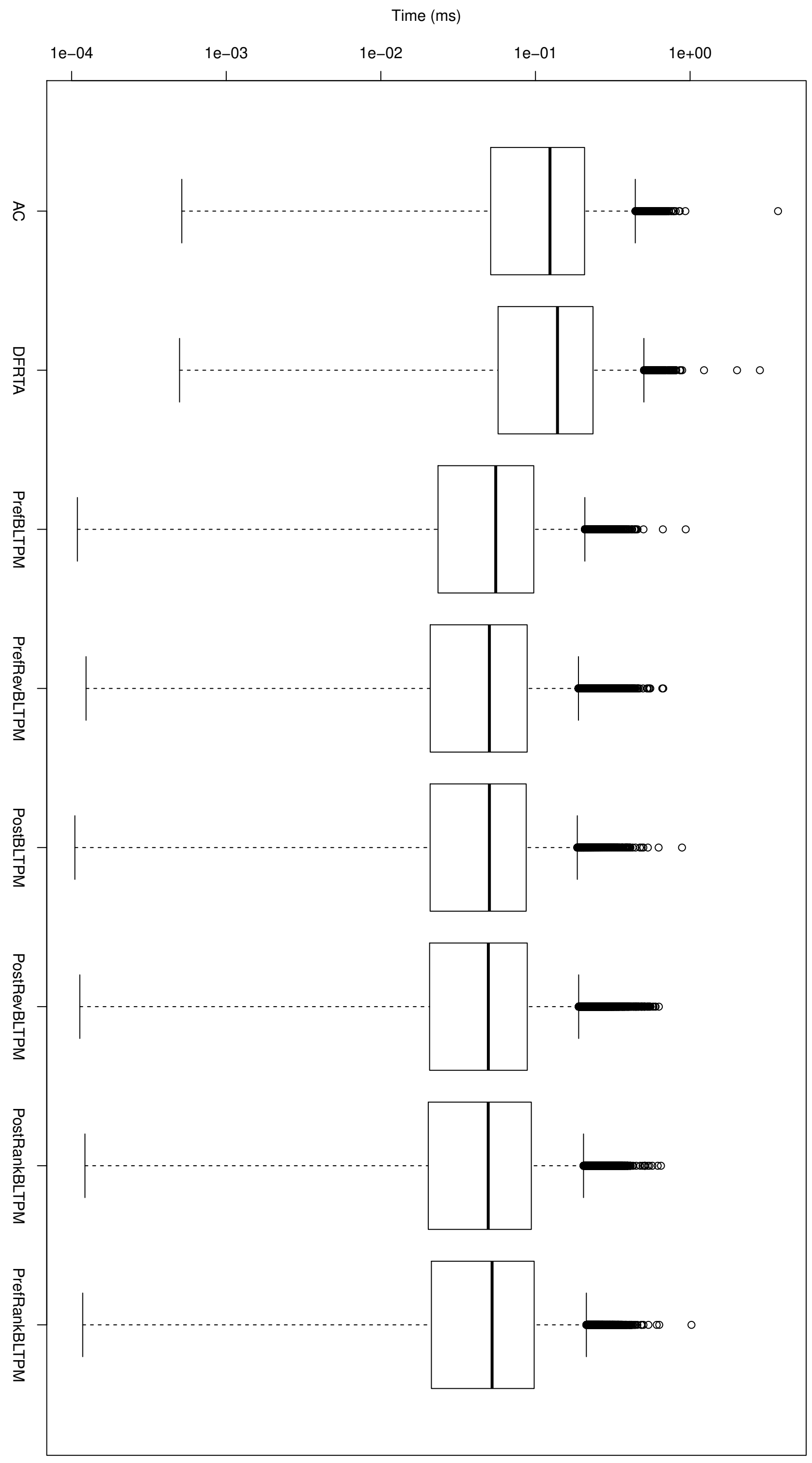


Figure 5.8: Distributions of pattern matching times for the respective algorithms on 150 trees of ca. 500 nodes each.

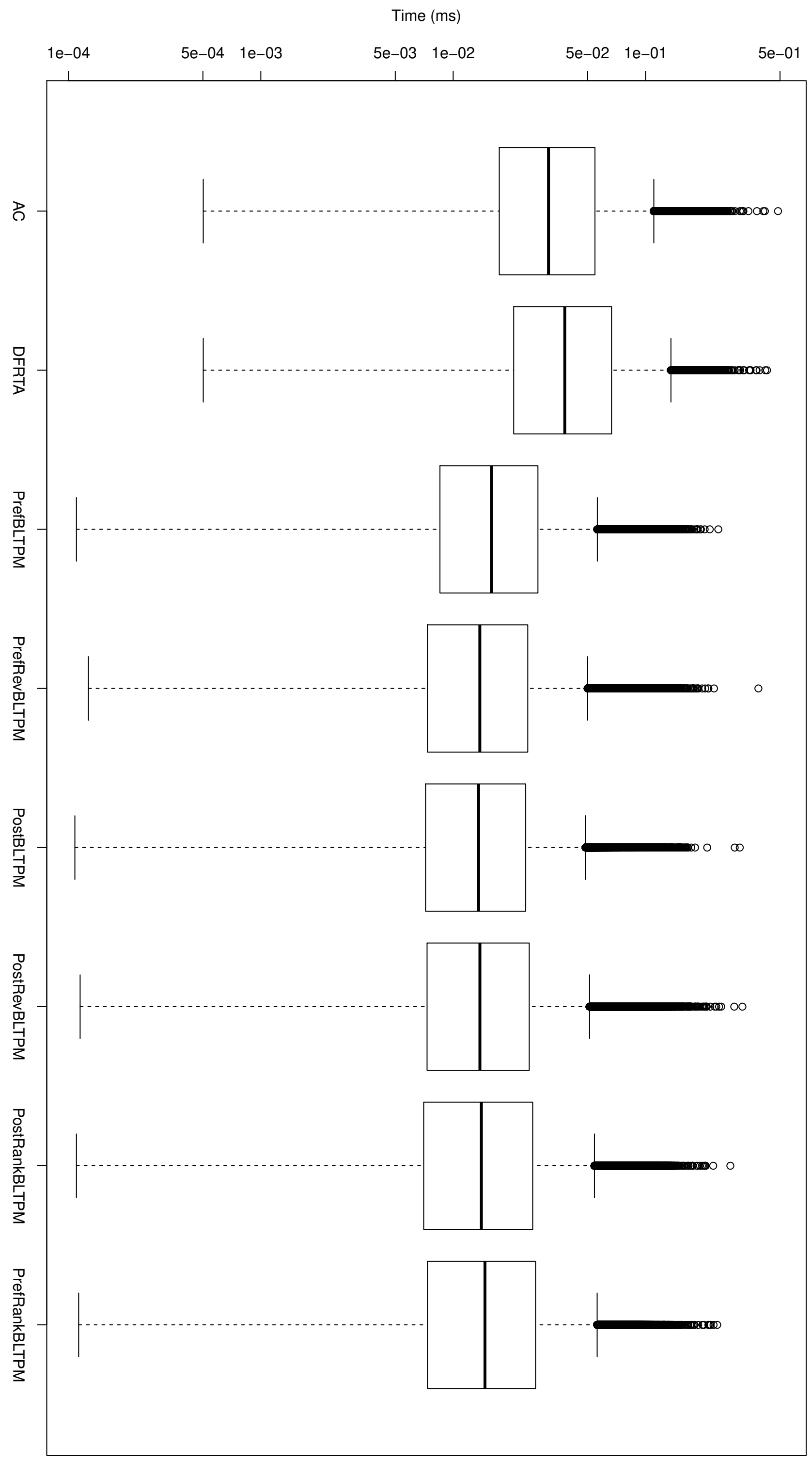


Figure 5.9: Distributions of pattern matching times for the respective algorithms on 500 trees of ca. 150 nodes each.

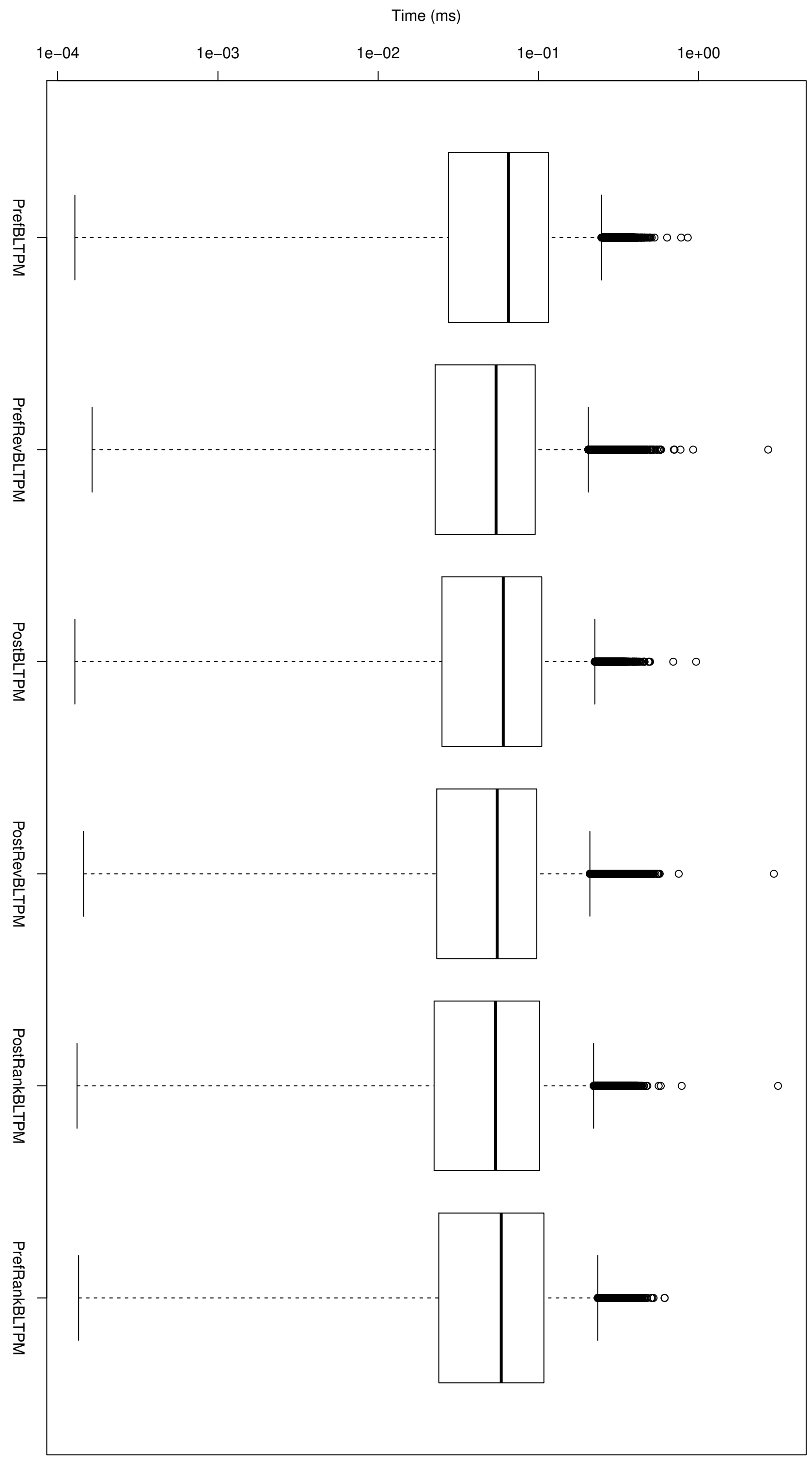


Figure 5.10: Distributions of nonlinear pattern matching times for the respective algorithms on 150 trees of ca. 500 nodes each.

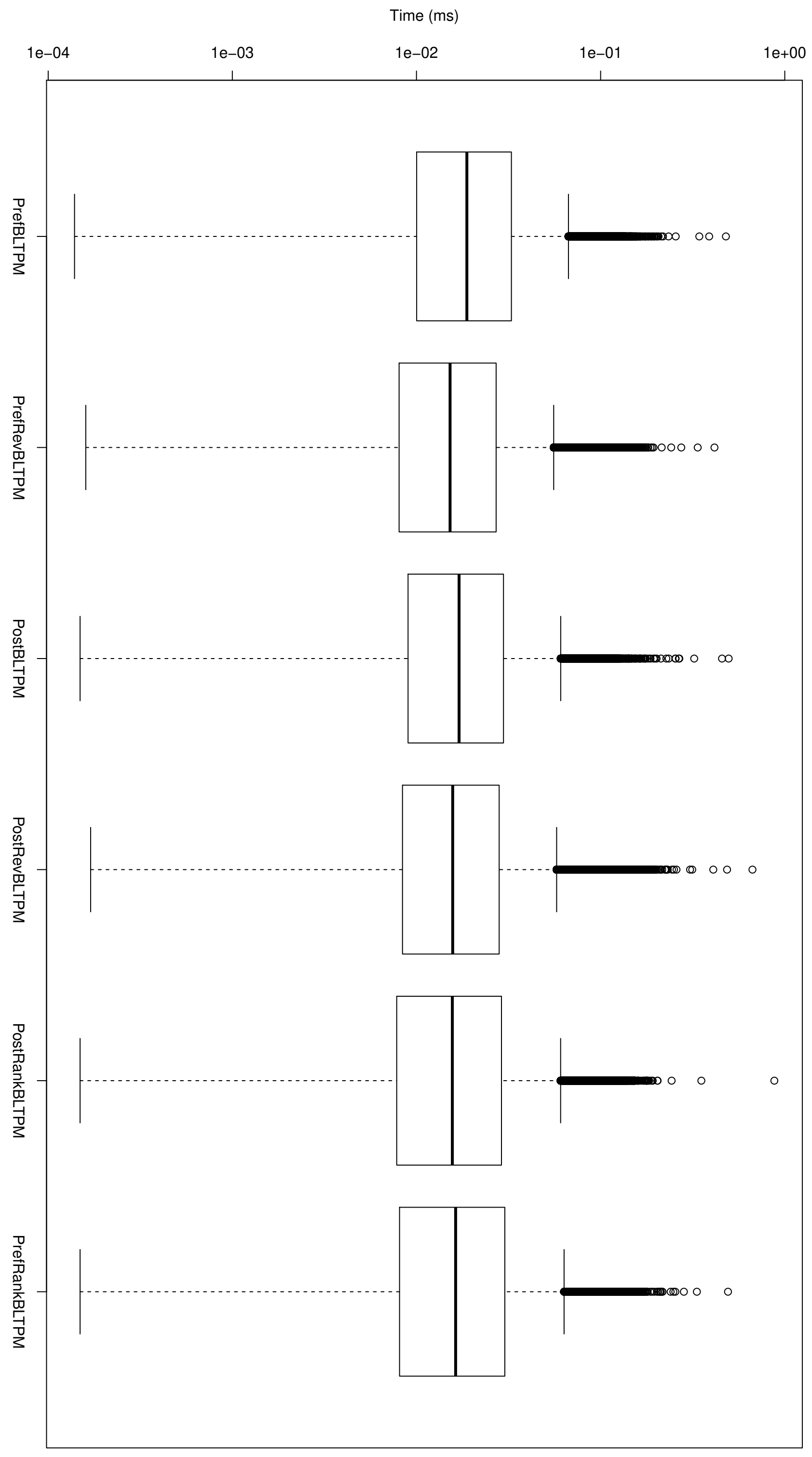


Figure 5.11: Distributions of nonlinear pattern matching times for the respective algorithms on 500 trees of ca. 150 nodes each.

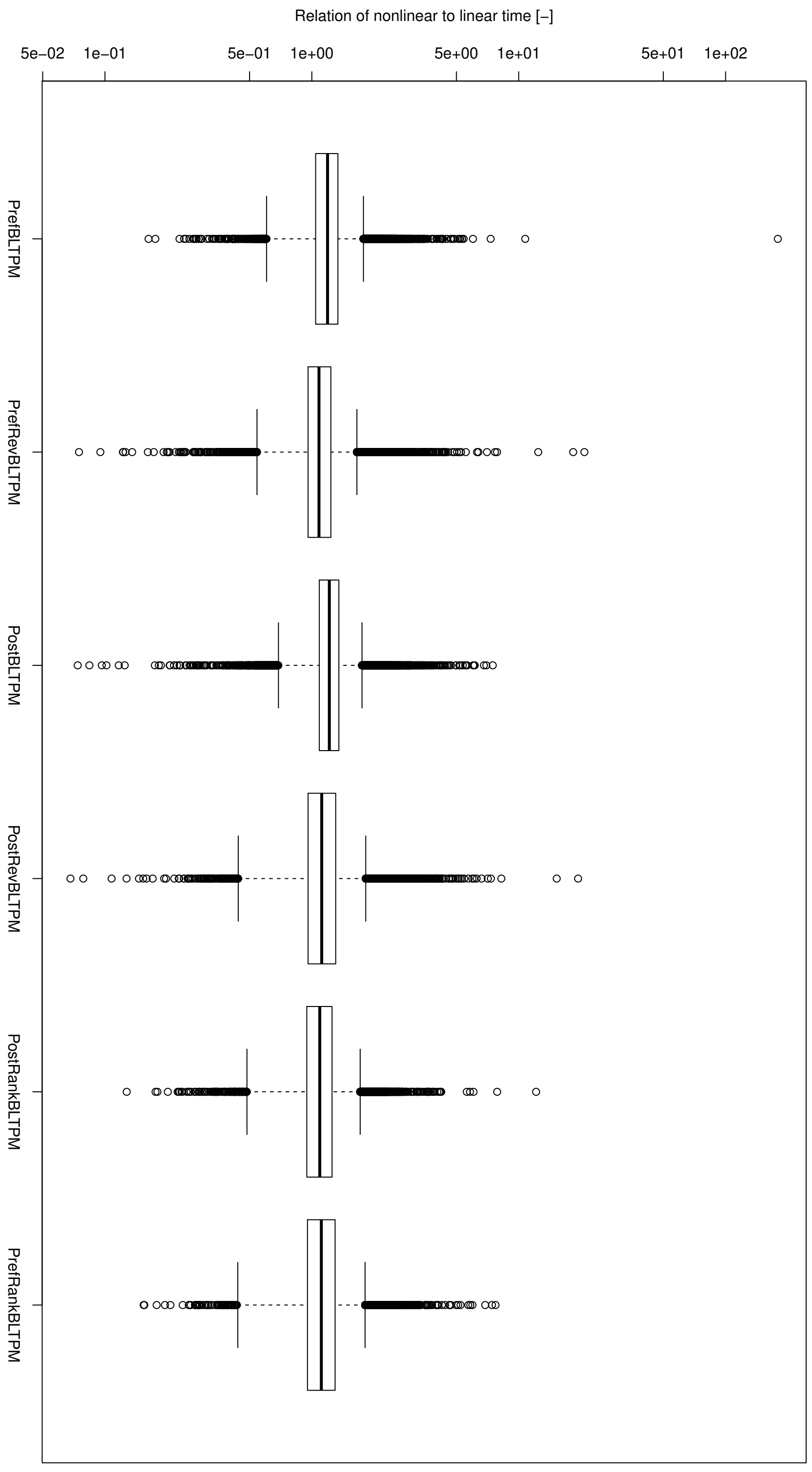


Figure 5.12: Relative slowdown of nonlinear pattern matching algorithm with respect to the pattern matching algorithm on 150 trees of ca. 500 nodes each.

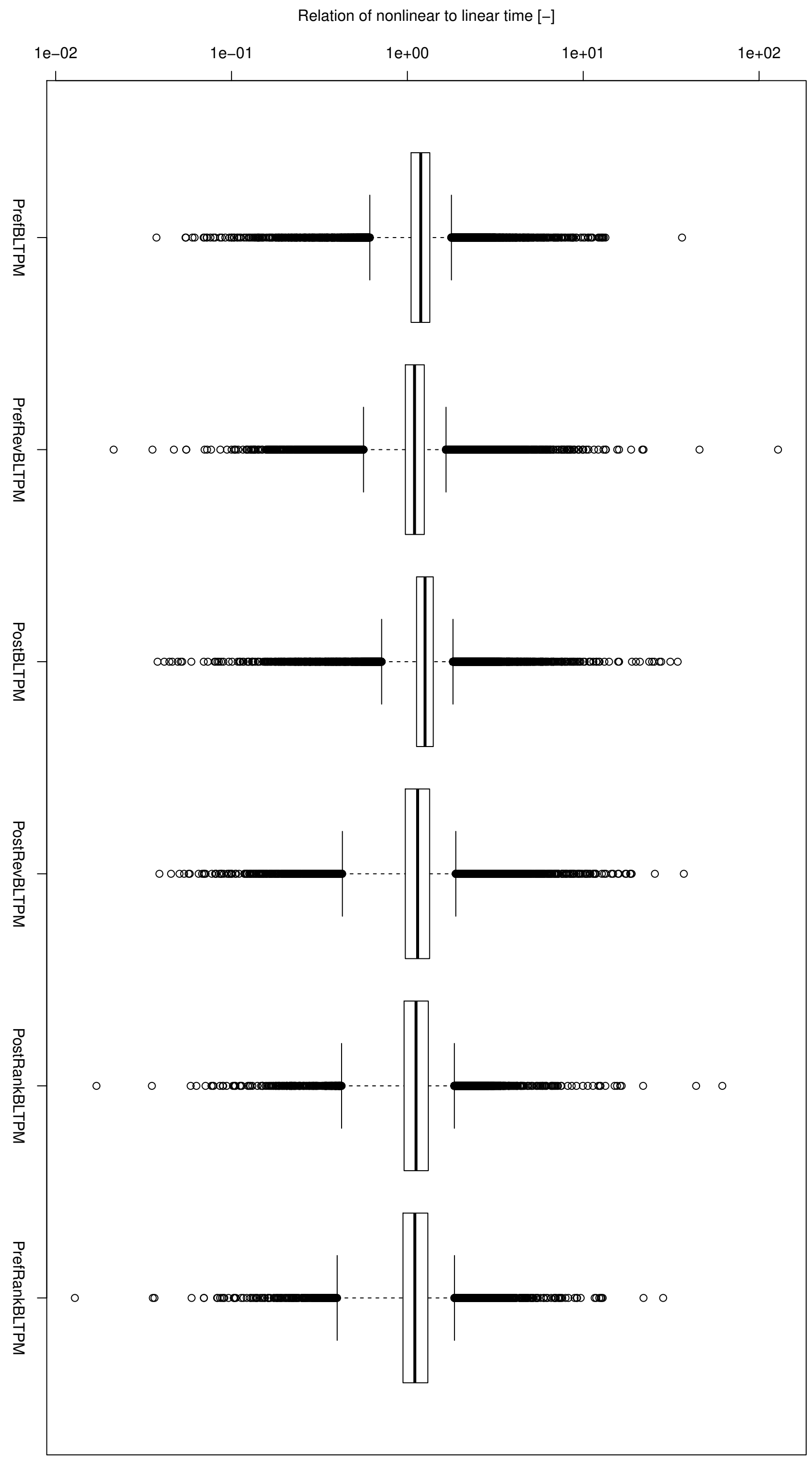


Figure 5.13: Relative slowdown of nonlinear pattern matching algorithm with respect to the pattern matching algorithm on 500 trees of ca. 150 nodes each.

Chapter 6

Conclusions

There are two main goals of the thesis, both related to (nonlinear) tree patterns. The first goal is to explore possibilities of indexing trees for (nonlinear) tree pattern matching. The second goal is related to (nonlinear) tree pattern matching.

6.1 Tree indexing conclusion

As the fulfilment of the first goal, the thesis introduces a pushdown automata approach to indexing trees for nonlinear tree pattern matching. The automata based index is presented in three variants, where the first is a basic approach, and second and third are space improved variants of the index. The thesis also presents a joint work on the linear space tree index as an alternative to indexing trees for tree pattern matching. The linear space tree index is extended so that it can report occurrences of nonlinear tree patterns as well. The presented indexes were implemented and experimentally evaluated.

A more detailed conclusion of tree indexing is presented as a last section of the respective tree indexing chapter.

The content of the tree indexing chapter was presented as a conference paper [62], as a journal paper [60], and as a conference paper [40].

6.2 Tree pattern matching conclusion

As the fulfilment of the second goal, the thesis introduces a backward linearised tree pattern matching algorithm. The backward linearised tree pattern matching algorithm is presented in more variants for various linear notations of trees, and its modification for nonlinear tree patterns is also presented. The presented tree pattern matching algorithm variants were implemented and experimentally evaluated.

A more detailed conclusion of tree pattern matching is presented as a last section of the respective tree indexing chapter.

The content of the tree pattern matching chapter was presented as a conference paper [61].

6.3 Future work suggestions

For the future work the following topics extending the presented results may be explored:

- Investigating possible compaction of tree pattern pushdown automata and its nonlinear extension to reduce the space complexity of the index they represent.
- Using a similar approach to the one used in the linear space tree index, i.e. splitting of the pattern at wildcard positions and connecting occurrences of these tree pattern parts, in the tree pattern matching. Instead of a string index, use a string pattern matching algorithm and again connect occurrences of the tree pattern parts.
- Creating a taxonomy of linearised tree pattern matching algorithms covering backward and forward approaches based on the results from this thesis, the results of adaptation of other shift heuristics for the backward pattern matching [8, 26] and results of adaptation of Knuth-Morris and dead-zone algorithms for linearised tree pattern matching [55].
- Investigating properties of trees, how they change properties of string pattern matching algorithms when used for searching for occurrences of subtrees in a subject tree. Design improvements of string pattern matching algorithms based on identified properties so that searching for subtrees is more efficient.

Bibliography

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53 – 86, 2004. The 9th International Symposium on String Processing and Information Retrieval.

[2] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, pages 491–516, 1989.

[3] A. V. Aho and J. D. Ullman. *The theory of parsing, translation, and compiling.* Prentice-Hall, 1972.

[4] P. Bille, I. L. Gortz, H. W. Vildhoj, and S. Vind. String indexing for patterns with wildcards. In F. Fomin and P. Kaski, editors, *Algorithm Theory - SWAT 2012*, volume 7357 of *Lecture Notes in Computer Science*, pages 283–294. Springer Berlin Heidelberg, 2012.

[5] P. Bille, I. Li Gørtz, H. W. Vildhøj, and D. K. 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.

[6] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.-T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[7] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[8] K. Červený. Návrh a implementace modifikací algoritmu protisměrného vyhledávání ve stromech. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2018.

[9] C. Charras and T. Lecroq. *Handbook of exact string matching algorithms.* King's College Publications, 2004.

[10] D. R. Chase. An improvement to bottom-up tree pattern matching. In *POPL*, pages 168–177. ACM Press, 1987.

[11] M. Christou, M. Crochemore, T. Flouri, C. S. Iliopoulos, J. Janoušek, B. Melichar, and S. P. Pissis. Computing all subtree repeats in ordered trees. *Information Processing Letters*, 112(24):958 – 962, 2012.

[12] L. Cleophas. Forest FIRE and FIRE Wood: Tools for tree automata and tree algorithms. In J. Piskorski, B. W. Watson, and A. Yli-Jyrä, editors, *FSMNLP*, volume 19 of *Frontiers in Artificial Intelligence and Applications*, pages 191–198. IOS Press, 2008.

[13] L. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, Apr. 2008.

[14] E. G. Coffman, Jr. and J. Eve. File structures using hashing functions. *Commun. ACM*, 13(7):427–432, July 1970.

[15] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic o$(nlog^3n)$ time. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 245–254, 1999.

[16] B. Commentz-Walter. A string matching algorithm fast on the average. In *International Colloquium on Automata, Languages, and Programming*, pages 118–132. Springer, 1979.

[17] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata: Techniques and applications, 2007. `http://www.grappa.univ-lille3.fr/tata/`.

[18] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

[19] M. Crochemore and C. Hancart. Automata for matching patterns. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 399–462. Springer Berlin Heidelberg, 1997.

[20] M. Crochemore and C. Hancart. Algorithms and theory of computation handbook. chapter Pattern Matching in Strings, pages 13–13. Chapman & Hall/CRC, 2010.

[21] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings.* Cambridge Univ Pr, 2007.

[22] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[23] M. Crochemore and W. Rytter. *Jewels of stringology.* World Scientific, 2002.

[24] M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In A. Apostolico and J. Hein, editors, *CPM*, volume 1264 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 1997.

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

[26] M. Cvach. Quick search vyhledávání ve stromech. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2018.

[27] A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S.-W. Woo. Position heaps: A simple and dynamic text indexing data structure. *J. Discrete Algorithms*, 9(1):100–121, 2011.

[28] S. Faro and T. Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2):13, 2013.

[29] C. Ferdinand, H. Seidl, and R. Wilhelm. Tree automata for code selection. *Acta Inf.*, 31(9):741–760, Nov. 1994.

[30] T. Flouri, C. S. Iliopoulos, J. Janoušek, B. Melichar, and S. P. Pissis. Tree indexing by pushdown automata and subtree repeats. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 899–902, Sept 2011.

[31] T. Flouri, J. Janoušek, B. Melichar, C. S. Iliopoulos, and S. P. Pissis. Tree template matching in ranked ordered trees by pushdown automata. In B. Bouchou-Markhoff, P. Caron, J.-M. Champarnaud, and D. Maurel, editors, *Implementation and Application of Automata*, volume 6807 of *Lecture Notes in Computer Science*, pages 273–281. Springer Verlag, 2011.

[32] O. Gauwin and J. Niehren. Streamable fragments of forward xpath. In B. Bouchou-Markhoff, P. Caron, J.-M. Champarnaud, and D. Maurel, editors, *Implementation and Application of Automata*, volume 6807 of *Lecture Notes in Computer Science*, pages 3–15. Springer Berlin Heidelberg, 2011.

[33] F. Gécseg and M. Steinby. *Tree Languages*, volume 3 of *Handbook of Formal Languages*, pages 1–68. Springer, 1997.

[34] P. J. Hatcher and T. W. Christopher. High-quality code generation via bottom-up tree pattern matching. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 119–130, New York, NY, USA, 1986. ACM.

[35] C. Hemerik and J. Katoen. Bottom-up tree acceptors. *Science of Computer Programming*, 13(1):51 – 72, 1989.

[36] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[37] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Boston, 2nd edition, 2001.

[38] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.

[39] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. *Discrete Applied Mathematics*, 146(2):156 – 179, 2005. 12th Annual Symposium on Combinatorial Pattern Matching.

[40] J. Janoušek, B. Melichar, R. Polách, M. Poliak, and J. Trávníček. A full and linear index of a tree for tree patterns. In *Descriptional Complexity of Formal Systems*, pages 198–209. Springer, 2014.

[41] J. Janoušek. String suffix automata and subtree pushdown automata. In *Proceedings of the Prague Stringology Conference 2009*, pages 160–172. Czech Technical University in Prague, Prague, 2009.

[42] J. Janoušek. Tree indexing by deteministic automata. *Dagstuhl reports*, 3(5):6, 2013.

[43] J. Janoušek and B. Melichar. On regular tree languages and deterministic pushdown automata. *Acta Inf.*, 46(7):533–547, 2009.

[44] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Annual Symposium on Combinatorial Pattern Matching*, pages 186–199. Springer, 2003.

[45] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[46] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Annual Symposium on Combinatorial Pattern Matching*, pages 200–210. Springer, 2003.

[47] G. Kucherov. On-line construction of position heaps. *Journal of Discrete Algorithms*, 20:3 – 11, 2013. StringMasters 2011 Special Issue.

[48] H.-W. Lang, M. Schimmler, and H. Schmeck. Matching tree patterns sublinear on the average. Technical report, Christian-Albrechts-Universität, 1980.

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

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

[51] B. Melichar. Arbology: Trees and pushdown automata. In A.-H. Dediu, H. Fernau, and C. Martín-Vide, editors, *Language and Automata Theory and Applications*, volume 6031 of *Lecture Notes in Computer Science*, pages 32–49. Springer Berlin Heidelberg, 2010.

[52] B. Melichar, J. Janoušek, and T. Flouri. Arbology: Trees and pushdown automata. *Kybernetika*, 48(3):402–428, 2012.

[53] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, *Combinatorial Pattern Matching*, volume 1448 of *Lecture Notes in Computer Science*, pages 14–33. Springer Berlin Heidelberg, 1998.

[54] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings of the 2009 Data Compression Conference*, DCC '09, pages 193–202, Washington, DC, USA, 2009. IEEE Computer Society.

[55] R. Obůrka. Vyhledávání ve stromech na principu mrtvých zón. M.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2016.

[56] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), July 2007.

[57] I. V. R. R. Ramesh. Nonlinear pattern matching in trees. *Journal of the Association for Computing Machinery*, 39(2), 1992.

[58] W. F. Smyth. *Computing Patterns in Strings.* Addison-Wesley-Pearson Education Limited, 2003.

[59] R. Strolenberg. ForestFIRE & FIREWood, A Toolkit & GUI for Tree Algorithms. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, June 2007. `http://alexandria.tue.nl/extra1/afstversl/wsk-i/strolenberg2007.pdf`.

[60] J. Trávníček, J. Janoušek, and B. Melichar. Indexing ordered trees for (nonlinear) tree pattern matching by pushdown automata. *Computer Science and Information Systems*, 9(3):1125–1153, 2012.

[61] J. Trávníček, J. Janoušek, B. Melichar, and L. Cleophas. Backward linearised tree pattern matching. In *Language and Automata Theory and Applications*, pages 599–610. Springer, 2015.

[62] J. Trávníček, J. Janoušek, and B. Melichar. Indexing trees by pushdown automata for nonlinear tree pattern matching. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 871–878. IEEE, 2011.

[63] B. W. Watson. A Boyer-Moore (or Watson-Watson) type algorithm for regular tree pattern matching. In *Stringology*, pages 33–38, 1997.

Reviewed Publications of the Author Relevant to the Thesis

[1] J. Janoušek, B. Melichar, R. Polách, M. Poliak, and J. Trávníček. A full and linear index of a tree for tree patterns. In *Descriptional Complexity of Formal Systems*, pages 198–209. Springer, 2014.

[2] J. Trávníček, J. Janoušek, and B. Melichar. Indexing trees by pushdown automata for nonlinear tree pattern matching. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 871–878. IEEE, 2011.

[3] J. Trávníček, J. Janoušek, and B. Melichar. Indexing ordered trees for (nonlinear) tree pattern matching by pushdown automata. *Computer Science and Information Systems*, 9(3):1125–1153, 2012. (IF: 0.556)

[4] J. Trávníček, J. Janoušek, B. Melichar, and L. Cleophas. Backward linearised tree pattern matching. In *Language and Automata Theory and Applications*, pages 599–610. Springer, 2015.

The paper has been cited in:

- A. Belabbaci, H. Cherroun, L. Cleophas, and D. Ziadi. Tree pattern matching from regular tree expressions. *Kybernetika*, 54(2):221–242, 2018.
- J. El-Qurna, H. Yahyaoui, and M. Almulla. A new framework for the verification of service trust behaviors. *Knowledge-Based Systems*, 121:7–22, 2017.

Remaining Rewieved Publications of the Author

[1] R. Polách, J. Trávníček, J. Janoušek, and B. Melichar. A new algorithm for the determinisation of visibly pushdown automata. In *2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Lódz, Poland, September 13-16, 2015*, pages 915–922, 2015. (IF: 0.623)

[2] R. Polách, J. Trávníček, J. Janoušek, and B. Melichar. Efficient determinization of visibly and height-deterministic pushdown automata. *Computer Languages, Systems & Structures*, 46:91–105, 2016.

The paper has been cited in:

- M. Nosál', J. Porubän, and M. Sulír. Customizing host ide for non-programming users of pure embedded dsls: A case study. *Computer Languages, Systems & Structures*, 49:101–118, 2017.

[3] T. Pecka, J. Trávníček, R. Polách, and J. Janoušek. Construction of a pushdown automaton accepting a postfix notation of a tree language given by a regular tree expression. In *7th Symposium on Languages, Applications and Technologies, SLATE 2018, June 21-22, 2018, Guimaraes, Portugal*, pages 6:1–6:12, 2018.

Remaining Publications of the Author Relevant to the Thesis

[A.1] Trávníček, Jan. *Arbology - Nonlinear Tree Pattern Matching Pushdown Automata.* Ph.D. Minimum Thesis, Faculty of Information Technology, Prague, Czech Republic, 2010.

Appendix **A**

Proofs Omitted from the Main Text

This apendix chapter is present to provide proofs omitted from the main text.

A.1 Proofs of Tree Pattern Matching Variants

Theorem A.1.1. Consider Theorem 5.3.4: Given tree pattern p in the prefix ranked bar notation and reversed bad character shift table $RBCS\_pref\_ranked\_bar(p)$ constructed by Algorithm 25 (ConstructRBCS), Algorithm 26 (ReversedBackwardLTPM) correctly computes the locations of all occurrences of pattern p in input tree t.

Proof. The reversed backward linearised tree pattern matching algorithm for prefix ranked bar notation is a reverse of the backward linearised tree pattern matching algorithm for ranked bar notation. It is to be proved that shifting using the $RBCS\_pref\_ranked\_bar(p)$ cannot skip any occurrence of the tree pattern p. Let there be a symbol $c = pref\_ranked\_bar(t)[i]$, where i is the position of current match attempt and $c \in \mathcal{A}$. Assume that there is an occurrence of p located at position $i - shift$, $0 < shift < RBCS\_pref\_ranked\_bar(p)[c]$. A symbol c must then be located at some position $shift$ either directly or as a part of a subtree that corresponds to a wildcard S. According to Definition 5.3.1, the $RBCS\_pref\_ranked\_bar(p)[c]$ is derived from the first occurrence of symbol c in the prefix ranked bar notation of the pattern $pref\_ranked\_bar(p)$, hence we get a contradiction. The shift is also derived from the first occurrence of symbol $\uparrow S$ and its counterpart S in the prefix ranked bar notation of the pattern $pref\_ranked\_bar(p)$. If the symbol c is located in the subtree that corresponds to a wildcard S and its bar $\uparrow S$, then the shift is already computed from the smallest possible subtree containing the symbol c. Hence, pattern p cannot occur at position $shift$. Therefore, no occurrence of p can be skipped by the algorithm.

The algorithm also checks individual positions inside the subject tree for occurrences. The check is an extension of the same check performed by the backward string pattern matching algorithm. Individual symbols of the pattern tree other than the wildcard S and its bar $\uparrow S$ are compared with the subject tree in the same manner as the string version of the algorithm does. Wildcard S and its bar $\uparrow S$ are handled using table $SJT\_bar(t)$ which allows skipping subtrees of $pref\_ranked\_bar(t)$. $\square$

Theorem A.1.2. Consider Theorem 5.4.5: Given tree pattern p in the postfix ranked bar notation and postfix bad character shift table $BCS\_post\_ranked\_bar(p)$ constructed by Algorithm 27 (ConstructPBCS), Algorithm 28 (PostfixBackwardLTPM) correctly computes the locations of all occurrences of pattern p in subject tree t.

Proof. The backward linearised tree pattern matching algorithm for postfix ranked bar notation is similar to the reversed backward linearised tree pattern matching algorithm for prefix ranked bar notation. It is to be proved that shifting using the $BCS\_post\_ranked\_bar(p)$ cannot skip any occurrence of the tree pattern p. Let there be a symbol $c = post\_ranked\_bar(t)[i]$, where i is the position of current match attempt and $c \in \mathcal{A}$. Assume that there is an occurrence of p located at position $i - shift$, $0 < shift < BCS\_post\_ranked\_bar(p)[c]$. A symbol c must then be located at some position $shift$ either directly or as a part of a subtree that corresponds to a wildcard S. According to Definition 5.4.3, the $BCS\_post\_ranked\_bar(p)[c]$ is derived from the first occurrence of symbol c in the postfix ranked bar notation of the pattern $post\_ranked\_bar(p)$, hence we get a contradiction. The shift is also derived from the first occurrence of symbol S and its bar $\uparrow S$ in the postfix ranked bar notation of the pattern $post\_ranked\_bar(p)$. If the symbol c is located in the subtree that corresponds to a wildcard S and its bar $\uparrow S$, then the shift is already computed from the smallest possible subtree containing the symbol c. Hence, pattern p cannot occur at position $shift$. Therefore, no occurrence of p can be skipped by the algorithm.

The algorithm also checks individual positions inside the subject tree for occurrences. The check is an extension of the same check performed by the backward string pattern matching algorithm. Individual symbols of the pattern tree other than the wildcard S and its bar $\uparrow S$ are compared with the subject tree in the same manner as the string version of the algorithm does. Wildcard S and its bar $\uparrow S$ are handled using table $SJT\_bar(t)$ which allows skipping subtrees of $post\_ranked\_bar(t)$. $\square$

Theorem A.1.3. Consider Theorem 5.5.3: Given tree pattern p in the postfix ranked bar notation and reversed postfix bad character shift table $RBCS\_post\_ranked\_bar(p)$ constructed by Algorithm 29 (ConstructPRBCS), Algorithm 30 (PostfixReversedBackwardLTPM) correctly computes the locations of all occurrences of pattern p in input tree t.

Proof. The reversed backward linearised tree pattern matching algorithm for postfix ranked bar notation is a reverse of the backward linearised tree pattern matching algorithm for postfix ranked notation. It is to be proved that shifting using the $RBCS\_post\_ranked\_bar(p)$ cannot skip any occurrence of the tree pattern p. Let there be a symbol $c = post\_ranked\_bar(t)[i]$, where i is the position of current match attempt and $c \in \mathcal{A}$. Assume that there is an occurrence of p located at position $i + shift$, $0 < shift < RBCS\_post\_ranked\_bar(p)[c]$. A symbol c must then be located at some position $shift$

either directly or as a part of a subtree that corresponds to a wildcard S. According to Definition 5.5.1, the $RBCS\_post\_ranked\_bar(p)[c]$ is derived from the last occurrence of symbol c in the postfix ranked bar notation of the pattern $post\_ranked\_bar(p)$, hence we get a contradiction. The shift is also derived from the last occurrence of symbol $\uparrow S$ and its counterpart S in the postfix ranked bar notation of the pattern $post\_ranked\_bar(p)$. If the symbol c is located in the subtree that corresponds to a wildcard S and its bar $\uparrow S$, then the shift is already computed from the smallest possible subtree containing the symbol c. Hence, pattern p cannot occur at position $shift$. Therefore, no occurrence of p can be skipped by the algorithm.

The algorithm also checks individual positions inside the subject tree for occurrences. The check is an extension of the same check performed by the backward string pattern matching algorithm. Individual symbols of the pattern tree other than the wildcard S and its bar $\uparrow S$ are compared with the subject tree in the same manner as the string version of the algorithm does. Wildcard S and its bar $\uparrow S$ are handled using table $SJT\_bar(t)$ which allows skipping subtrees of $post\_ranked\_bar(t)$. □

Theorem A.1.4. Consider Theorem 5.6.5: Given a tree pattern p in the prefix notation and shift table $BCS\_pref\ (p)$ constructed by Algorithm 31 (ConstructBCS_pref), Algorithm 32 (PrefixRankedBackwardLTPM) correctly computes the locations of all occurrences of pattern p in subject tree t.

Proof. The backward linearised tree pattern matching algorithm for prefix notation is based on the reversed backward linearised tree pattern matching algorithm for prefix notation. It is to be proved that shifting using the $BCS\_pref(p)$ cannot skip any occurrence of the tree pattern p. Let there be a symbol $c = pref(t)[i]$, where i is the position of current match attempt and $c \in \mathcal{A}$. Assume that there is an occurrence of p located at position $i - shift$, $0 < shift < BCS\_pref(p)[c]$. A symbol c must then be located at some position $shift$ either directly or as a part of a subtree that corresponds to a wildcard S. According to Definition 5.6.2, the $BCS\_pref(p)[c]$ is derived from the first occurrence of symbol c in the prefix notation of the pattern $pref(p)$, hence we get a contradiction. The shift is also derived from the first occurrence of symbol S in the prefix notation of the pattern $pref(p)$ if the symbol c is located in the subtree that corresponds to a wildcard S. Hence, pattern p cannot occur at position $shift$. Therefore, no occurrence of p can be skipped by the algorithm.

The algorithm also checks individual positions inside the subject tree for occurrences. The check is an extension of the same check performed by the backward string pattern matching algorithm. Individual symbols of the pattern tree other than the wildcard S are compared with the subject tree in the same manner as the string version of the algorithm does. Wildcard S is handled using table $SJT\_pref(t)$ which allows skipping subtrees of $pref(t)$. □

Theorem A.1.5. Consider Theorem 5.6.12: Given a tree pattern p in the postfix notation and the shift table $BCS\_post(p)$ constructed by Algorithm 33 (ConstructBCS_post), Algorithm 35 (PostfixRankedBackwardLTPM) correctly computes the locations of all occurrences of pattern p in subject tree t.

Proof. The backward linearised tree pattern matching algorithm for postfix notation is based on the reversed backward linearised tree pattern matching algorithm for postfix ranked notation. It is to be proved that shifting using the $BCS\_post(p)$ cannot skip any occurrence of the tree pattern p. Let there be a symbol $c = post(t)[i]$, where i is the position of current match attempt and $c \in \mathcal{A}$. Assume that there is an occurrence of p located at position $i + shift$, $0 < shift < BCS\_post(p)[c]$. A symbol c must then be located at some position $shift$ either directly or as a part of a subtree that corresponds to a wildcard S. According to Definition 5.6.7, the $BCS\_post(p)[c]$ is derived from the last occurrence of symbol c in the postfix notation of the pattern $post(p)$, hence we get a contradiction. The shift is also derived from the last occurrence of symbol S in the postfix notation of the pattern $post(p)$. If the symbol c is located in the subtree that corresponds to a wildcard S. Hence, pattern p cannot occur at position $shift$. Therefore, no occurrence of p can be skipped by the algorithm.

The algorithm also checks individual positions inside the subject tree for occurrences. The check is an extension of the same check performed by the backward string pattern matching algorithm. Individual symbols of the pattern tree other than the wildcard S are compared with the subject tree in the same manner as the string version of the algorithm does. Wildcard S is handled using table $SJT\_post(t)$ which allows skipping subtrees of $post(t)$. □