Dissertation thesis

A STRING AUTOMATA APPROACH TO TREE PATTERN MATCHING AND INDEXING

Ing. Eliška Šestáková

Submitted to
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
Supervisor: doc. Ing. Jan Janoušek, Ph.D.
August 18, 2022

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Ing. Eliška Šestáková. All rights reserved.
This dissertation thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The dissertation thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this dissertation thesis: Šestáková Eliška. *A string automata approach to tree pattern matching and indexing.* Dissertation thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

Acknowledgments v

Abstract vi

Notation and conventions viii

Abbreviations x

1 Introduction 1
1.1 Aim of the dissertation thesis . . . 2
1.2 Contribution of the dissertation thesis . . . 3
1.3 Structure of the dissertation thesis . . . 4

2 Preliminaries 5
2.1 Alphabets, strings, languages, and grammars . . . 5
2.1.1 Aphabets . . . 5
2.1.2 Strings . . . 5
2.1.3 Languages . . . 6
2.1.4 Grammars . . . 6
2.2 Finite and pushdown automata . . . 7
2.2.1 Finite automata . . . 7
2.2.2 Pushdown automata . . . 11
2.3 Trees . . . 11
2.3.1 Operations on trees . . . 14
2.3.2 Tree parts . . . 23
2.4 Tree languages . . . 24
2.4.1 Prefix bar notation for ordered labeled trees . . . 25
2.4.2 Path notation for ordered labeled trees . . . 28
2.4.3 XML and XPath . . . 30

3 Tree pattern matching 33
3.1 Classification of tree pattern matching problems . . . 34
3.1.1 Structure of the pattern . . . 34
3.1.2 Nature of the pattern . . . 35
3.1.3 Integrity of the pattern . . . 38
3.1.4 Number of patterns . . . 39
3.1.5 Way of matching the pattern . . . 39
3.1.6 Exactness of matching the pattern . . . 39
3.2 Describing tree pattern matching problems using the classification . . . 40
3.3 Possible extensions to the classification . . . 40
3.4 Summary . . . 42

4 Previous results and related work **43**
4.1 Inexact pattern matching . . . 43
4.1.1 Strings . . . 43
4.1.2 Trees . . . 49
4.2 Indexing . . . 56
4.2.1 Strings . . . 56
4.2.2 Trees . . . 63

5 Main results in inexact tree pattern matching **67**
5.1 Constrained 1-degree edit distance . . . 68
5.2 Problem statement . . . 71
5.3 Automata approach . . . 72
5.4 1-degree matching automaton for the constrained simple 1-degree edit distance . 76
5.4.1 Deterministic automaton . . . 81
5.4.2 Simulation by dynamic programming . . . 83
5.5 1-degree matching automaton for the simple 1-degree edit distance . . . 88
5.5.1 Pushdown automaton . . . 90
5.5.2 Finite automaton with ε-transitions . . . 95
5.5.3 Simulation of the 1-degree matching ε-NFA by dynamic programming . . 100
5.6 1-degree matching automaton for the (constrained) 1-degree edit distance . . . 107
5.6.1 Constrained 1-degree edit distance . . . 107
5.6.2 1-degree edit distance . . . 110
5.7 Summary . . . 118

6 Main results in tree indexing **121**
6.1 Problem statement . . . 122
6.2 Automata approach . . . 124
6.3 Rootpath automaton . . . 127
6.4 Path automaton . . . 130
6.5 Fully-gapped rootpath automaton . . . 133
6.6 Gapped rootpath automaton . . . 136
6.7 Summary . . . 140

7 Conclusions **143**
7.1 Contributions of the dissertation thesis . . . 143
7.2 Future work . . . 146

A Reviewed publications of the author relevant to the dissertation thesis **159**

B Remaining publications of the author relevant to the dissertation thesis **161**

C Remaining publications of the author **163**

This dissertation thesis has taken its time. I am tremendously grateful to everyone who allowed for it and cheered me along the way.

I thank my supervisor, Jan Janoušek, for his support, kindness, and patience. You are the one who inspired me to choose theoretical computer science as my branch of study and later gave me the idea to apply for the doctoral study program.

I also express my gratitude to my co-workers: Bořivoj Melichar, Jan Trávníček, Jan Holub, Štěpán Plachý, Tomáš Pecka, and Ondřej Suchý for generously sharing their wisdom in the course of my research.

In addition, I am very grateful for the friendship of Karolina Hrnčiříková. To her, I would like to say—Thank you for dedicating many hours of your summer vacation to help me convert my hand-drawn images to LaTeX.

Last but not least, I thank Conor White-Sullivan and his team for the Roam Research note-taking software. It helped me extensively to complete this dissertation thesis. I also thank Sönke Ahrens for writing the book "How to take smart notes" and thus introducing me to the systematic approach to research called Zettelkasten.

Finally, no words will do justice to the role of my parents and my fiancé, Ondra, in my life. Still, I would like to say that I am ridiculously grateful for all your love and the many sacrifices you have made to allow me to write this dissertation thesis. I am blessed to have been born into such a wonderful family and to have such a loving life partner. Thank you for being a constant support. Thank you for keeping me sane. Thank you for pushing me to be the best version of myself. As an insignificant repayment, I dedicate my work to you.

Abstract

The problem of tree pattern matching can be defined as the search for all occurrences of a pattern in an input tree. Often, this problem is declared to be analogous to the problem of pattern matching in strings. One of the approaches used for string pattern matching involves basing algorithms on the formalism of finite automata, which have been shown to be intuitive, efficient, and elegant tools for understanding and solving many string pattern matching problems.

Motivated by intuitive and elegant solutions to various string pattern matching problems using the automata theory, this dissertation thesis focuses on exploring a string automata approach to the problem of pattern matching in ordered labeled trees. Specifically, we explore a string automata approach to the inexact (approximate) tree pattern matching and tree indexing for linear XPath-inspired queries.

Furthermore, this dissertation thesis deals with a classification of tree pattern matching problems. The reason for this is that although various variants of the tree pattern matching problem appear in the literature, no unified naming standard for the problem exists. As a result, tree pattern matching problems are known under several names, making the comparison of research results unnecessarily difficult.

In summary, this dissertation thesis makes three key contributions. First, we propose a classification of tree pattern matching problems for ordered labeled trees. This classification categorizes these problems according to six criteria: the structure of the pattern, the nature of the pattern, the integrity of the pattern, the number of patterns, the way of matching the pattern, and the exactness of matching the pattern. The benefit of this classification is the provision of a unified naming standard for various tree pattern matching problems.

Second, we present novel methods based on string automata for the problem of inexact tree pattern matching. To measure the similarity between trees, we use the 1-degree edit distance, where elementary operations consist of node relabeling, leaf insertion, and leaf deletion. As a solution, we present a pushdown automaton built for a string representation of a pattern tree $\mathcal{P}$ and a nonnegative integer k that represents the maximum 1-degree edit distance allowed. The automaton can locate all occurrences of $\mathcal{P}$ in any input tree with up to k errors. Moreover, we discuss that the pushdown automaton can be transformed into an equivalent finite automaton due to its restricted use of the pushdown store. The finite automaton can then be made deterministic to obtain a linear-time search algorithm. However, this approach comes with high space complexity. Thus, as an alternative approach, we present an algorithm based on dynamic programming, the most widely used approach for computing tree edit distance and solving inexact tree pattern matching problems. Moreover, we show that our dynamic programming algorithm is a simulator of the corresponding nondeterministic finite automaton.

Third, we present a systematic approach based on string automata to the problem of indexing trees for linear XPath-inspired queries. Specifically, we focus on three XPath constructs: node test for node labels, /-axis (child axis), and //-axis (descendant axis). All indexes are constructed using the same systematic approach as a finite automaton which is shown to be suitable computational model for this problem, as all the considered queries can be represented as linear trees (paths).

Keywords: tree pattern matching, subtree matching, approximate tree pattern matching, inexact tree pattern matching, tree indexing, path indexing, XPath, 1-degree edit distance, finite automaton, pushdown automaton.

Abstrakt

Problém vyhledávání vzorků ve stromech lze definovat jako vyhledávání všech výskytů vzorku ve vstupním stromě. Tento problém je často připodobňován k problému vyhledávání v řetězcích. Jedním z přístupů používaném při vyhledávání v řetězcích jsou algoritmy založené na formalismu konečných automatů, které se ukázaly být intuitivním, elegantním a efektivním nástrojem pro pochopení a řešení mnoha problémů spojených s řetězci.

Intuitivnost a elegance automatového přístupu k problémům spojených s řetězci je motivací pro tuto disertační práci, která se zaměřuje na prozkoumání automatového přístupu k problému vyhledávání vzorků v uspořádaných stromech s pojmenovanými uzly. Konkrétně se tato práce zaměřuje na použití automatů (zpracovávajících řetězce) pro přibližné vyhledávání ve stromech a na indexování stromů pro lineární dotazy inspirované jazykem XPath.

Tato práce se dále zabývá klasifikací problémů vyhledávání ve stromech. Ačkoliv se totiž tyto problémy v literatuře vyskytují v různých variantách, nejsou pojmenovávány jednotně. Důsledkem toho je, že problémy vyhledávání ve stromech jsou známé pod různými názvy, což činí porovnání výsledků výzkumu zbytečně složité.

Tato práce má tři hlavní přínosy. Prvním je návrh klasifikace problémů vyhledávání vzorků v uspořádaných stromech s pojmenovanými uzly, a to podle šesti kritérií: struktura vzorku, povaha vzorku, celistvost vzorku, počet vzorků, způsob hledání výskytů a přesnost vyhledávání. Výhodou této klasifikace je, že umožňuje jednotné pojmenování různých problémů vyhledávání ve stromech.

Druhým přínosem této práce je představení nových metod pro přibližné vyhledávání vzorků ve stromech založených na automatech pro zpracování řetězců. Pro měření podobnosti stromů je použita jednostupňová (Selkowova) editační vzdálenost, jejíž základní editační operace jsou: přejmenování uzlu, vložení listu a smazání listu. Jako řešení představujeme zásobníkový automat zkonstruovaný pro lineární zápis zadaného stromového vzorku $\mathcal{P}$ a nezáporné celé číslo k s významem maximální dovolené editační vzdálenosti. Automat najde všechny výskyty vzorku $\mathcal{P}$ s nejvýše k chybami v jakémkoli vstupním stromu. Navíc je ukázáno, že zmíněný zásobníkový automat lze převést na ekvivalentní konečný automat díky omezenému použití zásobníku. Konečný automat lze posléze převést na deterministický a získat tím algoritmus na vyhledávání v lineárním čase. Bohužel tento přístup znamená vysokou paměťovou složitost. Proto jako alternativní přístup představujeme algoritmus založený na dynamickém programování, což je nejrozšířenější přístup pro výpočet stromové editační vzálenosti a pro řešení problémů přibližného vyhledávání ve stromech. Navíc je ukázáno, že náš algoritmus s dynamickým programováním je simulátorem nedeterministického konečného automatu.

Jako třetí přínos této práce je představený systematický automatový přístup k řešení problému indexování stromů pro lineární dotazy inspirované jazykem XPath. Konkrétně se v práci zaměřujeme na tři konstrukty z XPath: test na uzel na název elementu, /-osu (osa dětí) a //-osu (osa potomků). Všechny indexy jsou zkonstruované za použití stejného systematického přístupu jako konečné automaty. Ty se ukazují býti vhodným modelem výpočtu pro tento problém, neboť všechny uvažované dotazy mohou být reprezentované jako lineární stromy (cesty).

Klíčová slova: vyhledávání vzorků ve stromech, vyhledávání podstromů, přibližné vyhledávání vzorků ve stromech, indexování stromů, indexování cest, XPath, jednostupňová editační vzdálenost, konečný automat, zásobníkový automat.

Notation and conventions

Typesetting mathematics. In general, the following notation and font styles are used in this dissertation thesis for typesetting mathematics:

$A, \ldots, Z$	upper-case letters for sets and nonterminal symbols of a grammar
a, b, c	lower-case italic letters a, b, c for arbitrary set elements
i, j, k, l, m, n	lower-case italic letters i, j, k, l, m, n for nonnegative integer variables
p, q	lower-case italic letters p, q for states
u, v, w	lower-case italic letters u, v, w for vertices (nodes)
$\mathtt{a}, \ldots, \mathtt{z}$	typewriter font for specific alphabet symbols
$\boldsymbol{a}, \ldots, \boldsymbol{z}$	lower-case bold italic letters for sequences of symbols (i.e., strings)
$\boldsymbol{A}, \ldots, \boldsymbol{Z}$	upper-case bold italic letters for arrays and stacks
$\mathcal{A}, \ldots, \mathcal{Z}$	upper-case letters typeset with calligraphic font for composite objects; namely, graphs, trees, automata, and grammars
$\mathsf{prefBar}$	sans serif font and camel case naming conventions for multi-letter function names
d, f, δ, γ	lower-case italic letters d and f or lowercase Greek letters δ and γ for single-letter function names

Naming conventions. Moreover, the following general naming conventions often used in the literature are also adhered in this dissertation thesis:

L	for a language
$\mathcal{M}$	for an automaton
Σ	for an alphabet
Γ	for a pushdown alphabet
$\perp$	for the start (pushdown) symbol
Q	for a set of states
F	for a set of final states
δ	for an automaton transition function
ε	for the empty sequence of symbols chosen from an alphabet (i.e., the empty string)
$\boldsymbol{t}$	for an input string (text)
$\boldsymbol{p}$	for a pattern string
$\mathcal{G}$	for a graph and a grammar
$\mathcal{T}$	for an input tree
$\mathcal{P}$	for a pattern tree
V	for a set of vertices (nodes)
E	for a set of edges

Names are also sometimes used with a subscript ($\boldsymbol{x}_1$), superscript $(\boldsymbol{x}^1)$, prime symbol ($\boldsymbol{x}'$), or circumflex/hat ($\hat{\boldsymbol{x}}$) attached.

Numbers. All number variables used in this dissertation thesis are nonnegative integers. Thus, for brevity, it is not mentioned explicitly. For example, $i \geq 2$ is used instead of $i \in \mathbb{N} \wedge i \geq 2$. The set of natural numbers $\{1, 2, 3, \dots\}$ is denoted by $\mathbb{N}$. The set of natural numbers including zero is denoted by $\mathbb{N}_0$; that is $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

Sets. Given sets A and B, the relation that A is *subset* (possibly equal) of B is denoted by $A \subseteq B$. The *powerset* of A is denoted by 2^A. A function f from A to B is denoted by $f : A \to B$. By convention, every function is assumed to be total unless indicated otherwise.

Presentation of algorithms. To make the specifications of algorithms clear and easy to read, a presentation style suggested by Zobel [1, Chapter 10] called *prosecode* is used in this dissertation thesis. Thus, algorithms are described by text with embedded code rather than as code with textual annotations that may be unnecessarily difficult to read.

Each presentation of algorithms consists of a preamble and a body. The preamble contains specifications for the input and output and sometimes a brief explanation of the algorithm and algorithm-specific notation used in the body. In general, the body consists of code statements mixed with explanatory text with the following conventions:

1, 1a	Instead of using block-bounding statements such as `begin` and `end`, the algorithm's structure is given by numbered lists in which loops are presented as sublists with nested numbering.
$i \leftarrow 0$	Symbol "$\leftarrow$" denotes an assignment. For example, setting variable i to zero is written as set $i \leftarrow 0$.
$a = b$	The comparison of a and b (whether they are equal or not) is denoted by $a = b$. This comparison is true if a is equal to b. Otherwise, it is false.
$\boldsymbol{x}_i$	Mathematical notation is used instead of programming notation. For example, $\boldsymbol{x}_i$ is used rather $\boldsymbol{x}[i]$.

Abbreviations

DASG	Directed Acyclic Subsequence Graph
DAWG	Directed Acyclic Word Graph
DFA	Deterministic Finite Automaton
DTD	Document Type Definition
NFA	Nondeterministic Finite Automaton
PDA	Pushdown Automaton
SNINWE	Structure Nature Integrity Number Way Exactness
XML	Extensible Markup Language
XPath	XML Path Language

Chapter 1
Introduction

The problem of tree pattern matching can be defined as the search for all occurrences of a pattern in an input tree. This problem has applications in various areas, such as genetics [2], [3], code generation [4], [5], term rewriting [6], and XML (Extensible Markup Language) processing [7], [8].

In the literature, tree pattern matching has several variants that differ in the type of tree considered, the definition of the pattern, and the specification when a pattern is considered to occur in an input tree. A tree can, for example, be free or rooted, ordered or unordered, and labeled or unlabeled. A pattern can be described by an ordinary tree, a finite set of trees, or a tree where nodes are labeled by special symbols such as wildcards and variables. It can also be described as a regular tree expression or a query using the syntax of query language, such as XPath (XML Path Language). A pattern can be considered to occur in an input tree if, for example, a part of the input tree is the same as the pattern, or the matching process can be more tolerant and allow some errors.

Tree pattern matching can be considered as an extension of string pattern matching. One of the approaches used for string pattern matching is to base algorithms on the formalism of finite automata [9]–[15], which have been shown to be intuitive, efficient, and elegant tools for understanding and solving many string pattern matching problems. Moreover, the use of deterministic finite automata leads to linear-time searching algorithms. Automata-based methods also represent a systematic approach to solving string pattern matching problems and can be considered a unifying view of algorithms for string pattern matching [9], [12]. As a result, these algorithms are easier to understand, implement, and extend to solve similar problems. Some existing algorithms for string processing have also been shown to be simulators of nondeterministic finite automata that can be constructed to solve these tasks [12], [16]–[18].

In general, finite automata can be used in two ways to solve string pattern matching problems depending on whether the pattern or the input text changes more often. In case of a fixed pattern, efficiency is reached by preparing the pattern and creating a string matching automaton [10]. When looking for occurrences of different patterns in a fixed input text, it is suitable to preprocess the input text and create a data structure known as index. Examples of automata-based indexes are suffix or factor automata [19], [20].

Motivated by intuitive and elegant solutions to various string pattern matching problems using the automata theory, researchers have been exploring the adaptation of the automata approach to the domain of trees. One way to process a tree structure is to use a computational model known as tree automaton [21], [22], which works directly on the tree structure and can be viewed as the generalization of a string automaton. Another way is to encode trees as strings using linear notation and use string automata for their processing.

In 2008, Flouri, Janoušek, and Melichar [23] founded a new algorithmic discipline representing a systematic approach to tree pattern matching using string automata in the same way that string

automata are used as a unified strategy for string pattern matching. Since *stringology* is a popular nickname for string algorithms, Flouri, Janoušek, and Melichar called their approach *arbology* [23]–[25] from the Spanish word *árbol*, meaning tree. As its computational model, arbology uses a pushdown automaton that reads the linear notation of a tree and processes the underlying tree structure using the pushdown store. One of the main results of arbology research is that a deterministic pushdown automaton can solve any problem that can be solved by a finite tree automaton [26]. Among other results of arbology belong methods for indexing trees [23], [27]–[33], finding repeats of tree patterns in trees [23], [30], tree compression [34], [35], and searching subtrees [23], [30] and patterns with subtree wildcards [30], [36] in trees.

This dissertation thesis follows in the footsteps of arbology. We[1] propose systematic, elegant, and intuitive methods based on string automata for tree pattern matching problems that have not yet been addressed in arbology. We specify these problems in the following two sections where we introduce the aims and contributions of this dissertation thesis. As a conclusion of this chapter, we provide an overview of the structure of this dissertation thesis.

1.1 Aim of the dissertation thesis

In this dissertation thesis, we focus on pattern matching in rooted ordered labeled trees in which one node is distinguished from others, sibling order matters, and each node is associated with a label.

Although various variants of the tree pattern matching problem appear in the literature, no unified naming standard for the problem exists. As a result, tree pattern matching problems are known under several names, making the comparison of research results unnecessarily complex. For example, in its most simplified form, the input of the tree pattern matching problem is two trees, an input tree and a pattern tree, and the goal is to find all exact occurrences of the pattern tree in the input tree. This problem is known under names such as exact tree matching [8], subtree matching [24], [30], [33], [37], [38], or subtree isomorphism [39, Section 4.2], [40]. A variant of this problem allows the leaves of the pattern tree to be labeled by special symbols that match any subtree of the input tree. In the literature, these symbols are known as wildcards [22, Section 3.1.2], (sub)tree placeholders [6], or don't care symbols [30], [36], [41]. The corresponding problem is called tree pattern matching [6], [33] or (tree) template matching [30], [41]. The absence of a unified naming standard for tree pattern matching problems takes us to the first research question that we aim to answer in this dissertation thesis.

▶ **Question 1.1.** *How can various problems of tree pattern matching be presented together and in a common style such that their relations become clear?*

After solving the first question, our next aim is to further develop arbology research by proposing a systematic approach based on string automata to tree pattern matching problems that have not yet been addressed in arbology research. In particular, we are interested in two problems: inexact (approximate) tree pattern matching and XML processing. Both of these problems have been suggested as topics for future arbology research [25], [30].

The problem of inexact tree pattern matching is a variant of tree pattern matching that aims to find occurrences of the pattern in the input tree with up to a given maximum number of errors. This more tolerant type of tree pattern matching is useful if the pattern, input tree, or both can be subjects of deformation or corruption and the data cannot be corrected beforehand. Measuring the similarity between two trees is known as the tree edit distance problem (or tree-to-tree correction problem). It was first introduced by Selkow [42] in the late 70's as a generalization of

[1] By convention, the personal pronoun "we" is used throughout this dissertation thesis, although it is the work of a single author. Examiners may be interested in Appendix A containing a list of reviewed publications of the author of this dissertation thesis and a description of contributions the author has made in publications co-authored with researchers other than the supervisor.

the well-known string edit distance problem [43]. Since then, other types of tree edit distances have been introduced [44, Chapter 11], [45].

The problem of inexact tree pattern matching is a direct extension of inexact string pattern matching for which the automata approach has been studied before [12, Section 2.2.3.2], [9], [16], [46]–[48]. However, to our knowledge, although the problem of inexact tree pattern matching has been addressed [49]–[52], no existing solutions are based on the formalism of string automata. This leads to our second research question.

► **Question 1.2.** *How can existing automata-based methods for solving inexact string pattern matching be adapted to the inexact tree pattern matching problem?*

The second area in which we aim to develop arbology research further is inspired by XML processing. We aim to explore a systematic string automata approach to the problem of tree indexing for linear XPath-inspired queries.

► **Question 1.3.** *How can automata-based principles used for text indexing be adapted to the problem of indexing trees for linear XPath-inspired queries?*

These three questions are addressed in Chapters 3, 5, and 6.

1.2 Contribution of the dissertation thesis

This dissertation thesis makes three main contributions to the problem of tree pattern matching. Specifically, the contributions correspond to the aims described in the previous section.

The first contribution is a novel method of classification of tree pattern matching problems for ordered labeled trees called the SNINWE (Structure Nature Integrity Number Way Exactness) classification. It categorizes tree pattern matching problems according to six criteria: the structure of the pattern, the nature of the pattern, the integrity of the pattern, the number of patterns, the way of matching the pattern, and the exactness of matching the pattern. This classification benefits by providing a unified naming standard for various tree pattern matching problems.

The remaining two contributions reside in the adaptation of automata-based methods and principles used for inexact string pattern matching and text indexing to the domain of trees. These contributions also help to further develop arbology research. Moreover, we show that a finite automaton is a sufficient computational model for the problems we consider.

Our second contribution presents novel methods based on string automata for inexact tree pattern matching under the 1-degree edit distance. The 1-degree edit distance is an edit distance for ordered labeled trees introduced by Selkow [42], where elementary operations consist of node relabeling, leaf insertion, and leaf deletion. We consider both the unit cost and the non-unit cost variant of this distance. In the first case, computing the distance between two trees corresponds to seeking the minimum number of operations required to make the first tree isomorphic to the other. In the second case, we look for the cost of a least-cost sequence of operations. As a solution, we present a pushdown automaton built for a string representation of a pattern tree $\mathcal{P}$ and a nonnegative integer k that represents the maximum 1-degree edit distance allowed. The automaton can locate all occurrences of $\mathcal{P}$ in any input tree with up to k errors.

Moreover, we discuss that the pushdown automaton can be transformed into an equivalent finite automaton due to its restricted use of the pushdown store. The finite automaton can then be made deterministic to obtain a linear-time search algorithm. However, this approach comes with high space complexity. Thus, as an alternative approach, we propose a simulation of the nondeterministic finite automaton based on dynamic programming.

Our third contribution is an extension of arbology research to the domain of XML processing. We present a systematic approach based on string automata to the problem of indexing trees for linear XPath-inspired queries. Specifically, we focus on three XPath constructs: node test for node labels, /-axis (child axis), and //-axis (descendant axis).

First, we present a *rootpath automaton* that is an index of all the rootpaths (root-to-node paths) of an ordered labeled tree. Searching for rootpaths in a tree corresponds to the evaluation of XPath queries that consist only of the /-axis and the node test for node labels. Then, we introduce a *path automaton*, which is an index of all the paths of an ordered labeled tree. Searching for paths in a tree is analogous to evaluating XPath queries that start with the //-axis and continue only with the /-axis. As a third indexing automaton, we present a *fully-gapped rootpath automaton* that is an index of an ordered labeled tree for every linear *fully-gapped* pattern tree for which there exists a matching rootpath in the indexed tree. A linear fully-gapped pattern tree is similar to a subsequence used in the string domain. In the context of XPath, this automaton can evaluate all queries that consist only of the //-axis and the node test for node labels. As a final index, we introduce a *gapped rootpath automaton* that is an index of an ordered labeled tree for every linear *gapped* pattern tree for which there exists a matching rootpath in the indexed tree. A linear gapped pattern tree is an ordered labeled tree that can contain path wildcards. This type of pattern is inspired by XPath queries that can contain both the /-axis and the //-axis.

All indexes are constructed using the same systematic approach as a finite automaton which is shown to be a suitable computational model for this problem, as all the considered queries can be represented as linear trees (paths). In other words, since the underlying tree structure of linear queries is simple and fixed, the pushdown store for its processing is not needed.

1.3 Structure of the dissertation thesis

This dissertation thesis consists of seven chapters, including this one. Chapter 2 provides the theoretical background. It contains definitions of basic concepts from the theory of formal string languages, including grammars and automata. In addition, the chapter also introduces trees and tree languages. In Chapter 3, we introduce our classification of tree pattern matching problems for ordered labeled trees. We also provide examples of using the classification as a unified naming standard for tree pattern matching problems and discuss some of its possible extensions. Chapter 4 provides an overview of existing methods related to the two algorithmic problems considered in this dissertation thesis. As we represent trees as strings, we also discuss relevant methods from the domain of strings. In Chapter 5, we propose novel methods based on string automata for solving the inexact tree pattern matching problem under the 1-degree edit distance. In Chapter 6, we present a systematic approach to constructing string automata indexes for linear XPath-inspired queries. Chapter 7 concludes the dissertation thesis. Apart from providing an overview of the main results and conclusions arrived at, a list of open problems suitable for future investigation is also discussed in the last chapter.

Chapter 2
Preliminaries

This chapter is devoted to the theoretical background of this dissertation thesis. First, we provide definitions of basic concepts from the theory of formal (string) languages, including grammars and automata. Then, trees and tree languages are introduced. In addition, this chapter includes definitions of tree parts used in tree pattern matching, such as subtree, bottom-up subtree, or rootpath. We also present some operations on trees and discuss how trees can be represented as strings. Finally, we briefly describe XML and its query language called XPath.

2.1 Alphabets, strings, languages, and grammars

In this section, we define basic concepts from the theory of formal (string) languages that pervade the theory of automata. These concepts include alphabets, strings, and languages. We also consider the characterization of languages based on the concept of grammar.

2.1.1 Aphabets

An *alphabet* is a finite nonempty set whose elements are called *symbols*. Conventionally, we use the symbol Σ for an alphabet. For $a \in \Sigma$, we define $\overline{\{a\}} = \Sigma \setminus \{a\}$. When comparing two symbols, we say that the symbols *match* if they are equal; otherwise, we say that they *mismatch*.

2.1.2 Strings

A *string* over Σ is a finite sequence of elements of Σ. For simplicity of notation, we write a string as the simple juxtaposition of symbols that compose it. For example, if $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, then acab is a string over Σ.

The *length* of a string $\boldsymbol{x}$ is the length of the sequence associated with $\boldsymbol{x}$. We use the standard notation $|\boldsymbol{x}|$ to denote the length of string $\boldsymbol{x}$. For example, $|\mathtt{acab}| = 4$. The sequence of zero length is called the *empty string* and is denoted by ε.

When $\boldsymbol{x}$ is a nonempty string, we use $\boldsymbol{x}_i$, where $i \in \{1, \ldots, |\boldsymbol{x}|\}$, to denote the symbol at index (position) i of $\boldsymbol{x}$ with the convention that indices begin with 1. For example, if $\boldsymbol{x} = \mathtt{acba}$, then $\boldsymbol{x}_1 = \boldsymbol{x}_4 = \mathtt{a}, \boldsymbol{x}_2 = \mathtt{c}$, and $\boldsymbol{x}_3 = \mathtt{b}$.

Given two strings $\boldsymbol{x}$ and $\boldsymbol{y}$, we say that $\boldsymbol{x} = \boldsymbol{y}$ if $|\boldsymbol{x}| = |\boldsymbol{y}|$ and $\boldsymbol{x}_i = \boldsymbol{y}_i$ for each $i \in \{1, \ldots, |\boldsymbol{x}|\}$.

The *concatenation* of string $\boldsymbol{x}$ and string $\boldsymbol{y}$, denoted by $\boldsymbol{xy}$ or $\boldsymbol{x} \cdot \boldsymbol{y}$, is the string of length $|\boldsymbol{x}| + |\boldsymbol{y}|$ such that $\boldsymbol{xy} = \boldsymbol{x}_1\boldsymbol{x}_2 \ldots \boldsymbol{x}_{|\boldsymbol{x}|}\boldsymbol{y}_1\boldsymbol{y}_2 \ldots \boldsymbol{y}_{|\boldsymbol{y}|}$. In other words, the concatenation of two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is the string composed of the symbols of $\boldsymbol{x}$ followed by the symbols of $\boldsymbol{y}$. The concatenation operation is associative, and the neutral element of the concatenation is ε.

A string $\boldsymbol{x}$ is a *substring* (*factor*) of string $\boldsymbol{y}$ if there exist strings $\boldsymbol{w}$ and $\boldsymbol{z}$ such that $\boldsymbol{y} = \boldsymbol{wxz}$. For $\boldsymbol{w} = \varepsilon$, $\boldsymbol{x}$ is a *prefix* of $\boldsymbol{y}$; and for $\boldsymbol{z} = \varepsilon$, $\boldsymbol{x}$ is a *suffix* of $\boldsymbol{y}$. A substring is called *proper* if $\boldsymbol{x} \neq \boldsymbol{y}$. Analogously, we define a *proper prefix* and a *proper suffix*. Given a nonempty string $\boldsymbol{x}$, we denote its substring $\boldsymbol{x}_i\boldsymbol{x}_{i+1}\ldots\boldsymbol{x}_j$, where $1 \leq i \leq j \leq |\boldsymbol{x}|$, by $\boldsymbol{x}_{i\ldots j}$. A *subsequence* of $\boldsymbol{x}$ is a nonempty string obtained by deleting zero or more (not necessarily adjacent) symbols from $\boldsymbol{x}$.

Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be nonempty strings such that $\boldsymbol{x} = \boldsymbol{y}_{i\ldots j}$, where $1 \leq i \leq j \leq |\boldsymbol{y}|$. We say that $\boldsymbol{x}$ *occurs in* $\boldsymbol{y}$ or that there is an *occurrence* of $\boldsymbol{x}$ in $\boldsymbol{y}$. The position i is called the *start position* of the occurrence and the position j is called the *end position* of the occurrence.

When we *insert* string $\boldsymbol{y}$ into string $\boldsymbol{x}$ at position $i \in \{1, \ldots, |\boldsymbol{x}|\}$, string $\boldsymbol{x}_1 \ldots \boldsymbol{x}_{i-1}\boldsymbol{y}\boldsymbol{x}_i \ldots \boldsymbol{x}_{|\boldsymbol{x}|}$ is obtained. For example, if we insert string ab into string cddc at position 2, we get cabddc. Furthermore, inserting $\boldsymbol{y}$ into $\boldsymbol{x}$ at position 0 gives us string $\boldsymbol{yx}$, and inserting $\boldsymbol{y}$ into $\boldsymbol{x}$ at position $|\boldsymbol{x}| + 1$, gives us string $\boldsymbol{xy}$.

▶ **Definition 2.1** (Projection of a string onto a subalphabet)**.** Let Σ be an alphabet. Let $\Sigma' \subseteq \Sigma$. We define an infix operator $\downarrow$ that *projects* a string over Σ onto a string over Σ'. Formally, $\downarrow : \Sigma^* \times 2^\Sigma \to \Sigma^*$ is defined for $\boldsymbol{w} \in \Sigma^*$ as follows:

- $\varepsilon \downarrow \Sigma' = \varepsilon$ and
- $(a\boldsymbol{w}) \downarrow \Sigma' = \begin{cases} a(\boldsymbol{w} \downarrow \Sigma') & \text{if } a \in \Sigma', \\ \boldsymbol{w} \downarrow \Sigma' & \text{if } a \notin \Sigma'. \end{cases}$

2.1.3 Languages

Let Σ be an alphabet. We define Σ^k, where $k \geq 0$, to be the set of all strings over Σ of length k. For example, if $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, then $\Sigma^2 = \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$. Note that $\Sigma^0 = \{\varepsilon\}$, regardless of the symbols that alphabet Σ contains. Conventionally, *the set of all strings* over Σ is denoted Σ^*. Formally, $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$. Since $\Sigma^0 \subseteq \Sigma^*$, the empty string always belongs to Σ^*. Moreover, we use Σ^+ to denote *the set of all nonempty strings* over Σ; that is, $\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i$.

Let Σ be an alphabet. Then, $L \subseteq \Sigma^*$ is a *(string) language* over Σ. That is, a language is a set of strings all of which are chosen from Σ^*. A language L is called *finite* if it contains a finite number of strings. Otherwise, it is called *infinite*. For a finite language L, the number of its strings is denoted by $|L|$ and called the *size* of L. The *length* of language L is defined as $\sum_{\boldsymbol{x} \in L} |\boldsymbol{x}|$.

2.1.4 Grammars

A *grammar* is a quadruple $\mathcal{G} = (N, T, P, S)$, where N and T are finite disjoint sets (alphabets) of *nonterminal* and *terminal* symbols, respectively; $S \in N$ is the start symbol, and P is a set of production rules of the form $\alpha A \beta \to \gamma$, where $A \in N$ and $\alpha, \beta, \gamma \in (N \cup T)^*$. When the production rules are of the form $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$, the grammar is called *context-free*. When the production rules are of the form $A \to \alpha$, where $A \in N$ and $\alpha \in T^*N \cup T^*$, the grammar is called *regular*.

Given a grammar $\mathcal{G} = (N, T, P, S)$, we write $\alpha \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ instead of $\alpha \to \beta_1, \alpha \to \beta_2, \ldots, \alpha \to \beta_n$, where $\alpha \to \beta_1, \alpha \to \beta_2, \ldots, \alpha \to \beta_n$ are production rules from P.

Given a context-free or regular grammar $\mathcal{G} = (N, T, P, S)$ and $\boldsymbol{x}, \boldsymbol{y} \in (N \cup T)^*$, we say that $\boldsymbol{x}$ *derives* $\boldsymbol{y}$ *in one step*, denoted by $\boldsymbol{x} \Rightarrow \boldsymbol{y}$, if there exists $(A \to \alpha) \in P$ and $\beta, \gamma \in (N \cup T)^*$ such that $\boldsymbol{x} = \beta A \gamma$ and $\boldsymbol{y} = \beta \alpha \gamma$. Symbol $\Rightarrow^*$ is used for the *transitive and reflexive* closure of $\Rightarrow$. The *language* generated by $\mathcal{G}$, denoted by $L(\mathcal{G})$, is the set of strings $L(\mathcal{G}) = \{w : S \Rightarrow^* w \wedge w \in T^*\}$. A language that can be generated by a context-free grammar is called *context-free*. A language that can be generated by a regular grammar is called *regular*.

2.2 Finite and pushdown automata

In this section, we provide definitions of (string) automata that are computational models accepting (string) languages. In particular, we focus on a finite automaton and a pushdown automaton as a computational model recognizing regular languages and context-free languages, respectively. We also describe some basic algorithms for finite automata. A more detailed exposition of the (string) automata theory is given, for example, by Hopcroft et al. [53] or Kozen [54].

2.2.1 Finite automata

A finite automaton is a central notion in the regular string language theory. In this section, we define three types of finite automata and present three algorithms for finite automata.

▶ **Definition 2.2** (Finite automaton with ε-transitions, ε-NFA). A *finite automaton with ε-transitions* is a 5-tuple $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ such that

- Q is a finite set whose elements are called *states*;
- Σ is an alphabet called the *input alphabet*, and its elements are called *input symbols*;
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is a *transition function* whose elements are called *transitions*;
- $q_0 \in Q$ is the *start state*;
- $F \subseteq Q$ is a set whose elements are called *final states*.

For simplicity, we write $\delta(p, a)$ for $a \in \Sigma$ and $\delta(p, \varepsilon)$ instead of $\delta((p, a))$ and $\delta((p, \varepsilon))$, respectively. When $q \in \delta(p, a)$, we say that there is a transition from state p to state q labeled by a, or that if the automaton is in state p and reads input a, then it moves to state q. We also say that p has an *outgoing transition* to q labeled by a or that q has an *incoming transition* from p labeled by a. State p is the *source state* of the transition, a its *label*, and state q its *target state*. When $q \in \delta(p, \varepsilon)$, we say that there is ε-*transition* from state p to state q.

For an ε-NFA $(Q, \Sigma, \delta, q_0, F)$, function $\mathsf{eClose} : Q \to 2^Q$ called ε-*closure* is defined recursively, as follows:

- State q is in $\mathsf{eClose}(q)$.
- If $p \in \mathsf{eClose}(q)$, then all states contained in $\delta(p, \varepsilon)$ are in $\mathsf{eClose}(q)$. No other states are in $\mathsf{eClose}(q)$ than those produced from basis using the recursive step rule.

We apply the ε-closure to a set of states by taking the union of ε-closures for individual states. Formally, $\mathsf{eClose}(A) = \bigcup_{q \in A} \mathsf{eClose}(q)$, where A is a set of states.

Using the ε-closure, the *extended transition function* $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ is defined for an ε-NFA $(Q, \Sigma, \delta, q_0, F)$ and a string $\boldsymbol{w} \in \Sigma^*$ recursively, as follows:

$$\hat{\delta}(q, \boldsymbol{w}) = \begin{cases} \mathsf{eClose}(q) & \text{if } \boldsymbol{w} = \varepsilon, \\ \bigcup_{p_i \in \hat{\delta}(q, \boldsymbol{x})} \mathsf{eClose}(\delta(p_i, a)) & \text{if } \boldsymbol{w} = \boldsymbol{x}a \wedge a \in \Sigma. \end{cases}$$

In other words, $\hat{\delta}(q, \boldsymbol{w})$ is the set of states that can be reached along a sequence of transitions whose labels, when concatenated, form the string $\boldsymbol{w}$.

A state q of an ε-NFA $(Q, \Sigma, \delta, q_0, F)$ is *accessible* if there exists a string $\boldsymbol{w} \in \Sigma^*$ such that $q \in \hat{\delta}(q_0, \boldsymbol{w})$. A state that is not accessible is *unreachable*.

Given an ε-NFA $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$, we say that a string $\boldsymbol{w}$ is *accepted* by $\mathcal{M}$ (or that $\mathcal{M}$ *accepts* $\boldsymbol{w}$) if $\hat{\delta}(q_0, \boldsymbol{w}) \cap F \neq \emptyset$. Otherwise, $\boldsymbol{w}$ is *rejected* by $\mathcal{M}$. The set of all strings accepted by $\mathcal{M}$ is called the *language of* $\mathcal{M}$ (or the *language accepted by* $\mathcal{M}$) and is denoted by $L(\mathcal{M})$. Given

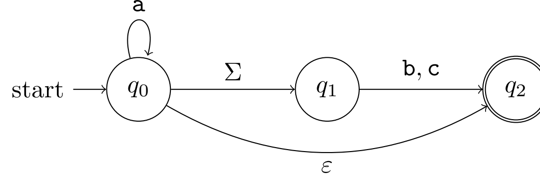


Figure 2.1 Pictorial representation of an ε-NFA $\mathcal{M} = (\{q_0, q_1, q_2\}, \Sigma, \{(q_0, \mathtt{a}, \{q_0, q_1\}), (q_0, \mathtt{b}, \{q_1\}), (q_0, \mathtt{c}, \{q_1\}), (q_0, \varepsilon, \{q_2\}), (q_1, \mathtt{a}, \emptyset), (q_1, \mathtt{b}, \{q_2\}), (q_1, \mathtt{c}, \{q_2\}), (q_1, \varepsilon, \emptyset), (q_2, \mathtt{a}, \emptyset), (q_2, \mathtt{b}, \emptyset), (q_2, \mathtt{c}, \emptyset), (q_2, \varepsilon, \emptyset)\}, q_0, \{q_2\})$, where $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Each state is illustrated as a circle. Final states are marked by a double circle while non-final states have a single circle. If $q \in \delta(p, a)$ for $a \in \Sigma \cup \{\varepsilon\}$, then we draw an arrow labeled by a that leads from the circle that corresponds to p to the circle that corresponds to q. If there are several input symbols that cause transitions from p to q, we use only one arrow labeled by the list of these symbols (or a set containing these symbols). We mark the start state q_0 by an arrow that does not originate at any state and leads to the circle that corresponds to q_0.

a language L, we say that $\mathcal{M}$ is *accepting* L if $L(\mathcal{M}) = L$. Two finite automata with ε-transitions are *equivalent* if they accept the same language. Given $q_f \in \hat{\delta}(q_0, \boldsymbol{w})$ and $q_f \in F$, we say that q_f is the *accepting state* for $\boldsymbol{w}$.

By restricting the transition function of an ε-NFA, we obtain a *nondeterministic* finite automaton (NFA) and *deterministic* finite automaton (DFA).

▶ **Definition 2.3** (Nondeterministic finite automaton, NFA)**.** A *nondeterministic finite automaton* is a 5-tuple $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ where all components have their same interpretation as for an ε-NFA, except that for each state $q \in Q$, it holds that $\delta(q, \varepsilon) = \emptyset$.

By convention, we simplify the transition function of an NFA to $\delta : Q \times \Sigma \to 2^Q$.

If we do not explicitly define $\delta(q, a)$ for $a \in \Sigma$ or $\delta(q, \varepsilon)$ for a state q in our algorithms that construct an NFA or ε-NFA, we assume that these transitions lead to the empty set.

By restricting the transition function of an NFA to be a function yielding a set of states that consists of at most one state, we obtain the deterministic finite automaton.

▶ **Definition 2.4** (Deterministic finite automaton, DFA)**.** A *deterministic finite automaton* is a 5-tuple $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ where all components have their same interpretation as for an NFA, except that for each pair $(q, a) \in Q \times \Sigma$, it holds that $|\delta(q, a)| \leq 1$.

For a DFA, it is natural to consider the transition function defined as a partial function $\delta : Q \times \Sigma \to Q$. In this dissertation thesis, we adopt this convention. We call a deterministic finite automaton $(Q, \Sigma, \delta, q_0, F)$ *total* when δ is defined for all pairs $(q, a) \in Q \times \Sigma$.

A DFA is called *(state) minimal* if there does not exist any equivalent DFA with less number of states. Any DFA can be converted into an equivalent minimal DFA; see Hopcroft et al. [53, Section 4.4] for details.

Conventionally, we describe a finite automaton using a *transition diagram*; see Figure 2.1.

2.2.1.1 Operations on finite automata

In this section, we describe three algorithms for finite automata. First, we provide an algorithm for eliminating ε-transitions. Second, we present an algorithm that transforms an NFA into an equivalent DFA. Then, we describe an algorithm that for given k deterministic finite automata $\mathcal{M}_1 \dots, \mathcal{M}_k$, returns a DFA accepting language $L(\mathcal{M}_1) \cup \dots \cup L(\mathcal{M}_k)$.

Given an ε-NFA, Algorithm 2.5 constructs an equivalent NFA. The set of states of the automaton remains the same. The NFA only differs in the set of final states and transitions. The computation of ε-closure of n states takes $\mathcal{O}(n^3)$ time [53, Section 4.3.1].

▶ **Algorithm 2.5** (Construction of an NFA equivalent to a given ε-NFA). Given an ε-NFA $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$, the algorithm constructs an NFA $\mathcal{M}' = (Q, \Sigma, \delta', q_0, F')$ that is equivalent to $\mathcal{M}$.

1. (Eliminate ε-transitions.) For each $q \in Q$ and for each $a \in \Sigma$: Set $\delta'(q, a) = \bigcup_{p \in \mathsf{eClose}(q)} \delta(p, a)$.
2. (Define the set of final states.) $F' = \{q : q \in Q \wedge \mathsf{eClose}(q) \cap F \neq \emptyset\}$.
3. (Return.) $\mathcal{M}' = (Q, \Sigma, \delta', q_0, F')$.

It is well known that every NFA can be turned into an equivalent DFA using so-called *subset* (or *powerset*) *construction* [55], [53, Section 2.3.5]. This algorithm involves constructing all subsets of the set of states of the NFA. These subsets are then considered to be the states of the corresponding DFA. Often, not all of these constructed states in the DFA are accessible. In Algorithm 2.6, we describe a variant of the subset construction that constructs only accessible states.

▶ **Algorithm 2.6** (Subset construction of a DFA equivalent to a given NFA). Given an NFA $\mathcal{M}_{\mathrm{N}} = (Q_{\mathrm{N}}, \Sigma, \delta_{\mathrm{N}}, q_{0\mathrm{N}}, F_{\mathrm{N}})$, the algorithm constructs a DFA $\mathcal{M}_{\mathrm{D}} = (Q, \Sigma, \delta, q_0, F)$ that is equivalent to $\mathcal{M}_{\mathrm{N}}$. During the construction, we use an attribute called *status* for each state in $\mathcal{M}_{\mathrm{D}}$. This attribute is either *marked* or *unmarked*.

1. (Initialize.)
   a. Set $q_0 \leftarrow \{q_{0\mathrm{N}}\}$.
   b. Set *status* of q_0 as unmarked.
   c. Set $Q \leftarrow \{q_0\}$.
2. (Determinization.) If *status* of each state in Q is marked, then continue with Step 3. Otherwise, choose an arbitrary state q from Q whose *status* is unmarked and execute the following steps:
   a. For each $a \in \Sigma$, set $\delta(q, a) \leftarrow \bigcup_{p \in q} \delta_{\mathrm{N}}(p, a)$.
   b. For each $a \in \Sigma$, set $Q \leftarrow Q \cup \{\delta(q, a)\}$.
   c. For each $a \in \Sigma$, set *status* of state $\delta(q, a)$ as unmarked.
   d. Set status of q as marked.
   e. Continue with Step 2.
3. (Define the set of final states.) $F = \{q : q \in Q \wedge q \cap F_{\mathrm{N}} \neq \emptyset\}$.
4. (Return.) Return $\mathcal{M}_{\mathrm{D}} = (Q, \Sigma, \delta, q_0, F)$.

Given an NFA with n states, the DFA constructed by Algorithm 2.6 has 2^n states in the worst case. For an arbitrary n, there is always an NFA for which there is no equivalent DFA with less than 2^n states [56]. The situation is different for some particular cases. See Theorem 2.7, Definition 2.8, and Theorem 2.9.

▶ **Theorem 2.7** (Salomaa et al. [57]). *Let Σ be an alphabet such that $|\Sigma| \geq 2$. Let $\mathcal{M}$ be an NFA with $n \geq 2$ states accepting a finite language over Σ. Then, the number of states of the DFA obtained from $\mathcal{M}$ by the subset construction is*

$$\mathcal{O}\left(|\Sigma|^{\frac{n}{\log|\Sigma|+1}}\right).$$

▶ **Definition 2.8** (Homogenous NFA [58]). Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be an NFA. For each $a \in \Sigma$, let $Q(a) = \{q : q \in \delta(p, a) \wedge p, q \in Q\}$. If for each pair of symbols $a, b \in \Sigma$, where $a \neq b$, it holds that $Q(a)$ and $Q(b)$ are disjoint sets, then $\mathcal{M}$ is called *homogenous*.

▶ **Theorem 2.9** (Champarnaud et al. [59], Melichar and Skryja [58]). *Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be a homogenous NFA. Then, the DFA obtained from $\mathcal{M}$ by the subset construction has $|Q'|$ states, where*

$$|Q'| \leq \sum_{a \in \Sigma} \left(2^{|Q(a)|} \right) - |\Sigma| + 1.$$

Let $\mathcal{M}_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $\mathcal{M}_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ be total deterministic finite automata. We can obtain a DFA accepting language $L(\mathcal{M}_1) \cup L(\mathcal{M}_2)$ using the standard algorithm based on *(Cartesian) product construction* [54, Page 22]. Let $\mathcal{M}_\cup = (Q_1 \times Q_2, \Sigma, \delta, (q_{01}, q_{02}), F)$, where

- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ for all $q_1 \in Q_1$, $q_2 \in Q_2$ and $a \in \Sigma$ and
- $F = \{(p, q) : p \in Q_1 \wedge q \in F_2\} \cup \{(p, q) : p \in F_1 \wedge q \in Q_2\}$.

Then, $L(\mathcal{M}_\cup) = L(\mathcal{M}_1) \cup L(\mathcal{M}_2)$ and $\mathcal{M}_\cup$ is deterministic.

Assuming that $|Q_1| = m$ and $|Q_2| = n$, automaton $\mathcal{M}_\cup$ has mn states. However, not all these states are always accessible. For example, if both $\mathcal{M}_1$ and $\mathcal{M}_2$ accept a finite language, then $mn - (m + n - 2)$ states are sufficient [60].

In this dissertation thesis, we use the Cartesian product construction for k deterministic finite automata (not necessarily total); see Algorithm 2.10. This algorithm constructs only accessible states. Assuming that each input DFA has (at most) n states, the resulting DFA has $\mathcal{O}(n^k)$ states.

▶ **Algorithm 2.10** (The product construction for k deterministic finite automata). Let $\mathcal{M}_1, \ldots \mathcal{M}_k$ be a sequence of $k \geq 1$ deterministic finite automata (not necessarily total) such as $\mathcal{M}_i = (Q_i, \Sigma, \delta_i, q_{0i}, F_i)$ for each $i \in \{1, \ldots, k\}$. The algorithm constructs a DFA $\mathcal{M}$ such that $L(\mathcal{M}) = L(\mathcal{M}_1) \cup \cdots \cup L(\mathcal{M}_k)$.

During the construction, we use an attribute called *status* for each state in $\mathcal{M}$. This attribute is either *marked* or *unmarked*.

1. (Initialize.)
   a. Set $q_0 \leftarrow (q_{01}, \ldots, q_{0k})$.
   b. Set *status* of q_0 as unmarked.
   c. Set $Q \leftarrow \{q_0\}$.
2. (Product construction.) If *status* of each state in Q is marked, then continue with Step 3. Otherwise, choose an arbitrary state $q = (q_1, \ldots, q_k)$ from Q whose *status* is unmarked and execute the following steps:
   a. For each $a \in \Sigma$, set $\delta(q, a) \leftarrow (p_1, \ldots p_k)$, where $\delta_i(q_i, a) = p_i$; if $\delta_i(q_i, a)$ is not defined or if $q_i = \emptyset_i$, set p_i to $\emptyset_i$.
   b. For each $a \in \Sigma$, set $Q \leftarrow Q \cup \{\delta(q, a)\}$.
   c. For each $a \in \Sigma$, set *status* of state $\delta(q, a)$ as unmarked.
   d. Set status of q as marked.
   e. Continue with Step 2.
3. (Define the set of final states.) $F = \{(q_1, \ldots, q_k) : (\exists q_i)(q_i \in F_i)\}$.
4. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

2.2.2 Pushdown automata

A pushdown automaton is a computational model accepting context-free languages.

▶ **Definition 2.11** (Pushdown automaton, PDA)**.** A *pushdown automaton* is a 7-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ such that

- Q is a finite set whose elements are called *states*;
- Σ is an alphabet whose elements are called *input symbols*;
- Γ is an alphabet called the *pushdown alphabet* whose elements are called stack symbols,
- δ is a mapping from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*$ into finite subsets of $Q \times \Gamma^*$.
- $q_0 \in Q$ is the *start state*;
- $\bot \in \Gamma$ is the *start symbol*;
- $F \subseteq Q$ whose elements are called *final states*.

Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ be a pushdown automaton. Triple $(q, \boldsymbol{w}, \alpha) \in Q \times \Sigma^* \times \Gamma^*$ is called a *configuration* of $\mathcal{M}$. We write the top of the pushdown store on its left hand side. The *initial* configuration of $\mathcal{M}$ is $(q_0, \boldsymbol{w}, \bot)$ for the input string $\boldsymbol{w} \in \Sigma^*$. The relation $\vdash_{\mathcal{M}} \subset (Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Sigma^* \times \Gamma^*)$ is a *transition* of $\mathcal{M}$. It holds that $(q, a\boldsymbol{w}, \alpha\beta) \vdash_{\mathcal{M}} (p, \boldsymbol{w}, \gamma\beta)$ if $(p, \gamma) \in \delta(q, a, \alpha)$. The k-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_{\mathcal{M}}$ is denoted $\vdash^k_{\mathcal{M}}, \vdash^+_{\mathcal{M}}, \vdash^*_{\mathcal{M}}$, respectively.

A language L accepted by a pushdown automaton $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ is defined in two distinct ways:

- Accepting by final state: $L(\mathcal{M}) = \{\boldsymbol{x} : (q_0, \boldsymbol{x}, \bot) \vdash^*_{\mathcal{M}} (q, \varepsilon, \gamma) \wedge \boldsymbol{x} \in \Sigma^* \wedge \gamma \in \Gamma^* \wedge q \in F\}$.
- Accepting by empty pushdown store: $L_\varepsilon(\mathcal{M}) = \{\boldsymbol{x} : (q_0, \boldsymbol{x}, \bot) \vdash^*_{\mathcal{M}} (q, \varepsilon, \varepsilon) \wedge \boldsymbol{x} \in \Sigma^* \wedge q \in Q\}$.

If the pushdown automaton accepts the language by empty pushdown store, then by convention we set F to be empty.

We describe pushdown automata using a transition diagram similarly as finite automata. The difference is that the arrows that represent individual transitions are now also labeled by pushdown operations. An arrow labeled a, α, β from state p to state q means that $(q, \beta) \in \delta(p, a, \alpha)$. Conventionally, we always use $\bot$ as the start symbol.

2.3 Trees

Several definitions of trees appear in the literature. For example, trees can be viewed as terms [21], [61], [54, Page 109] or defined based on the concepts of the graph theory [62, Section B.5], [63, Section 2.3.4] or using tree domain terminology, which was introduced by Gorn [64] in 1965 (as quoted by Gallier [65, Page 13] and Cleophas [22, Page 23]. In this dissertation thesis, we present trees using concepts from the graph theory. Thus, we first review the notion of a *graph*.

▶ **Definition 2.12** (Graph)**.** A *graph* $\mathcal{G}$ is a pair (V, E), where V and E are finite sets. The set V is called the *vertex set* of $\mathcal{G}$, and its elements are called *vertices*. The set of E is called the *edge set* of $\mathcal{G}$, and its elements are called *edges*. An edge $\{u, v\} \in E$ is a set of distinct vertices $u, v \in V$.

Let $\mathcal{G}$ be a graph. We use $V(\mathcal{G})$ and $E(\mathcal{G})$ to denote its vertex set and edge set, respectively. A *path* in $\mathcal{G}$ from a vertex u to a vertex u' is a nonempty sequence of distinct vertices $v_1, v_2, \ldots, v_{k-1}, v_k$ such that: (1) $v_1 = u$, (2) $v_k = u'$, and (3) $\{v_i, v_{i+1}\} \in E(\mathcal{G})$ for every $i \in \{1, \ldots, k-1\}$. If there is a path from u to u', we also say there is a u-u'-*path*. A vertex v

is *on path* $v_1, v_2, \ldots, v_{k-1}, v_k$ if there is $i \in \{1, \ldots, k\}$ such that $v_i = v$. The *length* of a path is defined as the number of edges in the path. For every vertex $v \in V(\mathcal{G})$, there is a 0-length path from v to v.

A graph $\mathcal{G}$ is *connected* if for every pair of vertices $u, v \in V(\mathcal{G})$ there is a path from u to v. For $k \geq 3$, a *cycle* in $\mathcal{G}$ is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such as there is an edge $\{v_i, v_{i+1}\} \in E(\mathcal{G})$ for every $i \in \{1, \ldots, k-1\}$ and $\{v_1, v_k\} \in E(\mathcal{G})$. A graph with no cycles is *acyclic*.

Based on the graph notion, we can now formally define the concept of a tree. In the literature, various tree types are used. Trees can, for example, be free or rooted, ordered or unordered, labeled or unlabeled, and ranked or unranked. The most general type of tree is a *free tree*, which is simply an acyclic connected graph. A free tree in which one node is distinguished from others is called a *rooted tree*. In this dissertation thesis, we consider all trees to be rooted, and our main focus is on ordered labeled trees. By convention, we call a vertex of a tree a *node*, and we say *node set* instead of vertex set.

▶ **Definition 2.13** (Rooted tree). A *rooted tree* is a triple $\mathcal{T} = (V, E, r)$, where (V, E) is an acyclic connected graph and $r \in V(\mathcal{T})$ is a node called the *root* of the tree.

Let $\mathcal{T} = (V, E, r)$ be a rooted tree. We use $|\mathcal{T}|$ to denote the *size* of $\mathcal{T}$, which we define as the number of nodes in the tree. It follows from Definition 2.13 that the size of a rooted tree is at least one since its node set always contains the root. Apart from r, we also often use $\mathsf{root}(\mathcal{T})$ to denote the root of $\mathcal{T}$. The edges of a rooted tree can be assigned a natural *orientation*, either *away from* or *towards* the root. Hence, rooted trees are in the literature also known as *oriented* trees. In this dissertation thesis, we consider edges in a rooted tree to be *undirected*, that is, without orientation. The alternative literature terminology used for rooted trees also includes the notion of an *unordered tree*. We often call a tree unordered instead of rooted to emphasize that the order of children is not significant.

Before formally defining ordered trees, we review some properties of a rooted tree and establish terminology that allows us to distinguish between the nodes of a rooted tree. Given a rooted tree $\mathcal{T}$, there exists a unique path from $\mathsf{root}(\mathcal{T})$ to every other node in $\mathcal{T}$. The length of a $\mathsf{root}(\mathcal{T})$-$v$-path, where $v \in V(\mathcal{T})$, is called the *depth* of node v, and it is denoted by $\mathsf{depth}(v)$. The *depth of tree* $\mathcal{T}$ is the maximum among the depths of all nodes in $\mathcal{T}$. A *level* of a rooted tree consists of all nodes at the same depth. For example, there is always only one node at level 0, which is the root of the tree. Furthermore, we can impose (vertical) partial ordering $\preceq_{\mathrm{V}}$ on $V(\mathcal{T})$ by letting $u \preceq_{\mathrm{V}} v$ for every pair of nodes $u, v \in V(\mathcal{T})$ if and only if u is on the $\mathsf{root}(\mathcal{T})$-$v$-path. Let $u, v \in V(\mathcal{T})$ be two distinct nodes such that $u \prec_{\mathrm{V}} v$, then node u is called an *ancestor* of v and node v is called a *descendant* of u. If node u is an ancestor of node v and $\{u, v\} \in E(\mathcal{T})$, then u is the *parent* of v and v is a *child* of u. Nodes with the same parent are called *siblings*. To reference the parent of node u, we use $\mathsf{parent}(u)$. The root is the only node without the parent. The number of children of node u is the *degree* of u, denoted by $\mathsf{degree}(u)$. We use $\mathsf{children}(u)$ to denote the set of nodes, each of which is a child of u. If $\mathsf{degree}(u) = 1$, we also use $\mathsf{child}(u)$ to denote the (only) child node of u. A node $u \in V(\mathcal{T})$ for which the set $\mathsf{children}(u)$ is empty is called a *leaf*. The set of all leaves in a rooted tree $\mathcal{T}$ is denoted by $\mathsf{leaves}(\mathcal{T})$. A node that is not a leaf is called an *internal node*. A rooted tree is called *linear* if its node set contains only one leaf. We use $\mathsf{leaf}(\mathcal{T})$ to denote the (only) leaf of a linear rooted tree $\mathcal{T}$.

Now, we can define the notion of an *ordered tree*. Informally, an ordered tree is a rooted tree, where the order of children for each node is significant.

▶ **Definition 2.14** (Ordered tree). An *ordered tree* is a quadruple $\mathcal{T} = (V, E, r, \preceq_{\mathrm{S}})$, where (V, E, r) is a rooted tree and $(V, \preceq_{\mathrm{S}})$ is a partially ordered set such that two distinct nodes are comparable in $\prec_{\mathrm{S}}$ if and only if they are siblings. The partial order relation $\preceq_{\mathrm{S}}$ is called the *sibling order*.

The terminology established for rooted trees can also be used for ordered trees. Moreover, we say that a node u of an ordered tree $\mathcal{T}$ *precedes* node v in $\mathcal{T}$ if $u \prec_{\mathrm{S}} v$.

For both ordered and unordered trees, we can choose to have labels on nodes, edges, or both. Our focus is on trees that have labels only on nodes. Thus, we omit the word "node" in the further text when referencing (un)ordered node labeled trees. We also say *labeled tree(s)* whenever the underlying trees can be ordered or unordered.

▶ **Definition 2.15** (Ordered labeled tree). Let Σ be an alphabet. An *ordered labeled tree* over Σ is a quintuple $\mathcal{T} = (V, E, r, \preceq_\mathrm{S}, \mathsf{label})$, where $(V, E, r, \preceq_\mathrm{S})$ is an ordered tree and $\mathsf{label} : V \to \Sigma$ is a *labeling function*. For every node $v \in V(\mathcal{T})$, the symbol $\mathsf{label}(v)$ is called a *label* of v.

Similarly, we define an *unordered labeled tree* as a quadruple $\mathcal{T} = (V, E, r, \mathsf{label})$ containing a labeling function label and the underlying rooted tree (V, E, r).

We now define a variant of a labeled tree in which labels restrict the number of children a node can have. For such trees, labels come from a *ranked alphabet*, where each symbol is associated with a nonnegative integer called *rank* (or *arity*).

▶ **Definition 2.16** (Ranked alphabet). Let Σ be an alphabet. A *ranked alphabet* is a pair (Σ, rank), where rank is a *ranking function* such that $\mathsf{rank} : \Sigma \to \mathbb{N}_0$. For every $a \in \Sigma$, the number $\mathsf{rank}(a)$ is called the *rank* of symbol a.

For a ranked alphabet (Σ, rank), we use Σ_k, where $k \geq 0$, to denote all symbols of Σ which rank is equal to k.

Ranked alphabets are used to define *ranked trees*, which are labeled trees, where labels come from a ranked alphabet and the degree of each node is determined by its label. Note that for ranked trees over a ranked alphabet (Σ, rank) to exist, the set Σ_0 should be nonempty.

▶ **Definition 2.17** (Ordered ranked tree). Let (Σ, rank) be a ranked alphabet. An *ordered ranked tree* over (Σ, rank) is an ordered labeled tree $\mathcal{T} = (V, E, r, \preceq_\mathrm{S}, \mathsf{label})$ over Σ such that for every node $v \in V$, it holds that $\mathsf{degree}(v) = \mathsf{rank}(\mathsf{label}(v))$.

Similarly, we define an *unordered ranked tree* as an unordered labeled tree $\mathcal{T}$ where the degree of every node $v \in V(\mathcal{T})$ is equal to the rank of the symbol $\mathsf{label}(v)$.

In addition to the notion of a ranked tree, we also say *unranked tree(s)* to denote labeled trees where the node labels do not determine the number of children. That is, an unranked tree can have two nodes with the same label but with a different number of children. Furthermore, we sometimes speak of trees without being explicit about orderedness, labeling, or rankedness. When this might confuse, we explicitly mention which specific tree type is meant.

By convention, we use $Tr(\Sigma)$ to denote the set of all ordered labeled trees over Σ. If Σ is equipped with a ranking function, then $Tr(\Sigma)$ denotes the set of all ordered ranked trees over (Σ, rank); otherwise, it denotes the set of all ordered unranked trees over Σ.

We often use a pictorial representation of ordered labeled trees; see Figure 2.2. Let $\mathcal{T}$ be an ordered labeled tree, then every node $v \in V(\mathcal{T})$ is represented as a circle, and each edge $\{u, v\} \in E(\mathcal{T})$ is represented as a line between u and v. The root of $\mathcal{T}$ is at the top, and other nodes are partitioned beneath it according to their depths. Thus, if for two distinct nodes $u, v \in V(\mathcal{T})$ we have $u \prec_\mathrm{V} v$, then the circle corresponding to node u appears higher in the drawing than the circle corresponding to node v. The horizontal order of nodes at each level of $\mathcal{T}$ is given by the sibling order relation: if for two distinct nodes $u, v \in V(\mathcal{T})$ we have $u \prec_\mathrm{S} v$, then the circle corresponding to u is drawn to the left of v. For every node $v \in V(\mathcal{T})$, its corresponding circle contains information about its label a and a unique *identifier* i in the form a^i. Identifiers allow us to easily reference individual nodes in the text since we introduce examples of ordered labeled trees only by its pictorial representation. Without loss of generality, we assign identifiers to nodes using the preorder numbering scheme, which is based on the *preorder tree traversal*[1] with the convention that the identifier of the root is equal to one.

[1] In the preorder traversal of a tree, the root is visited first, followed by the traversal of the bottom-up subtree rooted in turn at each of the children of the root. Specifically, the bottom-up subtree rooted at the first child is traversed first, followed by the bottom-up subtree rooted at the next sibling, etc. For precise definition, see, for example, Valiente [39, Section 3.1].

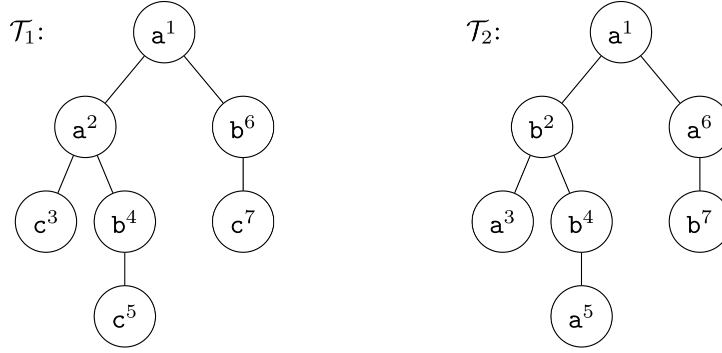


Figure 2.2 Pictorial representation of an ordered ranked tree $\mathcal{T}_1$ over a ranked alphabet $(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{(\mathtt{a}, 2), (\mathtt{b}, 1), (\mathtt{c}, 0)\})$ and an ordered unranked tree $\mathcal{T}_2$ over alphabet $\{\mathtt{a}, \mathtt{b}\}$. Nodes are numbered according to the order in which they are visited during the preorder traversal.

2.3.1 Operations on trees

In this section, we present some operations on ordered labeled trees. First, we define when two trees are considered to be the same tree using the concept of tree isomorphism. Then, we discuss how to measure similarity between two non-isomorphic trees using tree edit distance. Finally, we define different kinds of tree substitution.

2.3.1.1 Tree isomorphism

Isomorphism expresses when two trees are said to be the same tree. Informally, we consider two ordered labeled trees to be the same if they contain the same number of nodes connected and labeled in the same way.

▶ **Definition 2.18** (Tree isomophism). Let $\mathcal{T}_1 = (V_1, E_1, r_1, \preceq_{\mathrm{S}_1}, \mathsf{label}_1)$ and $\mathcal{T}_2 = (V_2, E_2, r_2, \preceq_{\mathrm{S}_2}, \mathsf{label}_2)$ be ordered labeled trees. A mapping $\varphi : V_1 \to V_2$ is a *tree isomorphism* of $\mathcal{T}_1$ and $\mathcal{T}_2$ if

- φ is a bijective mapping,
- $\{u, v\} \in E_1$ if and only if $\{\varphi(u), \varphi(v)\} \in E_2$ for every pair of nodes $u, v \in V_1$,
- $\varphi(r_1) = r_2$,
- $u \preceq_{\mathrm{S}_1} v$ if and only if $\varphi(u) \preceq_{\mathrm{S}_2} \varphi(v)$ for every pair of nodes $u, v \in V_1$, and
- $\mathsf{label}_1(u) = \mathsf{label}_2(\varphi(u))$ for every node $u \in V_1$.

We call two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ *isomorphic* if there is a tree isomorphism φ of $\mathcal{T}_1$ and $\mathcal{T}_2$. We denote this by $\mathcal{T}_1 \simeq \mathcal{T}_2$.

The problem of deciding whether two trees are isomorphic or not is called the *tree isomorphism problem* (or the *tree equivalence problem*). The tree isomorphism problem is closely related to the problem of editing trees since sometimes one of the trees (or both) can be subject to deformation or corruption, which means that we need to be more tolerant when comparing them. The problem of editing trees is called the *tree edit distance problem* (or the *tree-to-tree correction problem*), and we discuss it in the following section.

2.3.1.2 Tree edit distance

The tree edit distance problem measures the similarity between two trees. Informally, given two trees, we seek the minimum number (or cost) of changes required to make one tree isomorphic

to the other. This problem is a generalization of the well-known string edit distance problem; see Section 4.1.1. Similarly, as for strings, there are various types of tree edit distance based on the set of edit operations allowed [44, Chapter 15.4]. In this section, we describe a tree edit distance for ordered labeled trees considered by Selkow [42], where elementary operations consist of node relabeling, leaf insertion, and leaf deletion. Following a naming convention used in the literature [45], [44, Page 203], we call this distance the *1-degree edit distance.* We use this distance in Chapter 5, where we introduce our main results for inexact tree pattern matching. A brief overview of other edit distances for trees is given in Section 4.1.2, where we discuss previous results in inexact tree pattern matching.

Before we define the 1-degree edit distance, we formally specify its elementary edit operations.

▶ **Definition 2.19** (1-degree edit operations)**.** Let Σ be an alphabet, and let $\mathcal{T} = (V, E, r, \preceq_S, \mathsf{label})$ be an ordered labeled tree over Σ. A *1-degree edit operation* on $\mathcal{T}$ is either

- the *deletion* of a node $v \in \mathsf{leaves}(\mathcal{T}) \setminus \{r\}$, denoted by $\mathsf{del}(v)$;
- the *insertion* of a new leaf v labeled by $a \in \Sigma$ as the i-th child of a node $u \in V(\mathcal{T})$, where $i \in \{1, \ldots, \mathsf{degree}(u) + 1\}$, denoted by $\mathsf{ins}(a, i, u)$; or
- the *relabeling* of node $v \in V(\mathcal{T})$ into $a \in \Sigma \setminus \{\mathsf{label}(v)\}$ that sets $\mathsf{label}(v) = a$, denoted by $\mathsf{rel}(v, a)$.

The operation $\mathsf{del}(v)$ implies the deletion of the edge $\{\mathsf{parent}(v), v\}$ from E. Similarly, the operation $\mathsf{ins}(a, i, u)$ that inserts a new leaf v labeled by a implies insertion of an edge $\{u, v\}$ into E and including v in the sibling order $\preceq_S$ so that there are $i - 1$ children of u that percede v.

In other words, both deletion and insertion operations can be made only on leaves. Therefore, to delete an internal node v, for example, all descendants of v have to be deleted first so that v itself becomes a leaf. Even though insertion and deletion are restricted this way, any ordered labeled tree can be transformed into any other.

▶ **Theorem 2.20.** *Given a pair of ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$, there is always a sequence of 1-degree edit operations that makes $\mathcal{T}_1$ isomorphic to $\mathcal{T}_2$.*

Proof. This proof is based on the Valiente's proof [39, Lemma 2.2]. The difference is that Valiente allows deletion of the root. Let $\preceq_{\text{post}}$ be a total ordering imposed on $V(\mathcal{T}_1)$ by letting $u \prec_{\text{post}} v$ if and only if u is visited before v in the *postorder traversing*[2] of $\mathcal{T}_1$. Furthermore, $u \preceq_{\text{post}} v$ if and only if $u \prec_{\text{post}} v$ or $u = v$. Similarly, let $\preceq_{\text{pre}}$ be a total ordering imposed on $V(\mathcal{T}_2)$ by letting $u \prec_{\text{pre}} v$ if and only if u is visited before v in the preorder traversing of $\mathcal{T}_2$. Also, $u \preceq_{\text{pre}} v$ if and only if $u \prec_{\text{pre}} v$ or $u = v$. We can transform $\mathcal{T}_1$ into $\mathcal{T}_2$ as follows: (1) Delete all non-root nodes from $\mathcal{T}_1$ one-by-one in ascending order; that is, if $u \prec_{\text{post}} v$, delete u before deleting v. (2) If $\mathsf{label}(\mathsf{root}(\mathcal{T}_1)) \neq \mathsf{label}(\mathsf{root}(\mathcal{T}_2))$, then relabel $\mathsf{root}(\mathcal{T}_1)$ into $\mathsf{label}(\mathsf{root}(\mathcal{T}_2))$. (3) Insert all non-root nodes from $\mathcal{T}_2$ in ascending order into $\mathcal{T}_1$; that is, if $u \prec_{\text{pre}} v$, then insert u before inserting v. Inserting a node u means inserting a new leaf labeled by $\mathsf{label}(u)$ as a child of an appropriate node. ◀

Given ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$, a sequence of operations that makes $\mathcal{T}_1$ isomorphic to $\mathcal{T}_2$ is called an *edit script* between $\mathcal{T}_1$ and $\mathcal{T}_2$, denoted by $\mathsf{ed}(\mathcal{T}_1, \mathcal{T}_2)$, and we say that $\mathcal{T}_1$ *can be transformed into* $\mathcal{T}_2$ *via* $\mathsf{ed}(\mathcal{T}_1, \mathcal{T}_2)$. We define the *length* of an edit script as the number of operations in the script. See an example of an edit script in Figure 2.3.

A *cost* can accompany the 1-degree edit operations. In this dissertation thesis, we assume the cost of operations to be dependent on the labels involved and independent of the positions at

[2] In the postorder traversal of a tree, the bottom-up subtree rooted in turn at each of the children of the root is traversed first, and the root of the tree is visited last. Specifically, the bottom-up subtree rooted at the first child is traversed first, followed by the bottom-up subtree rooted at the next sibling, etc. For precise definition, see, for example, Valiente [39, Section 3.2].

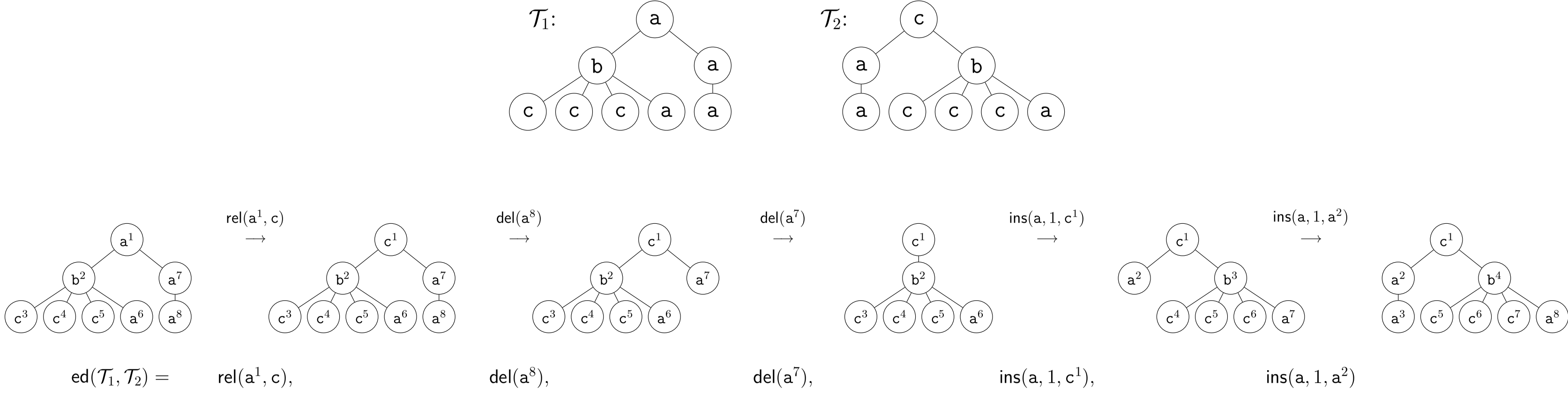


Figure 2.3 An example of two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ with an edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$. Note that the preorder numbers uniquely identifying each node that we use in the edit script do not always correspond to the node identifiers in the original tree $\mathcal{T}_1$. The preorder numbers change together with the structure change of $\mathcal{T}_1$.

	a	b	c	λ
a	0	1	4	9
b	1	0	3	8
c	4	3	0	5
λ	9	8	5	0

Table 2.1 An example of a cost function γ that is a metric on $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \cup \{\lambda\}$.

which operations take place. Before we define the cost of the operations, we define a *cost function* on an alphabet representing the set of possible labels in an ordered labeled tree. Conventionally, we extend the alphabet by a special symbol λ, called the *blank symbol*, that helps us later when assigning a cost to the insertion and deletion operations.

▶ **Definition 2.21** (Alphabet with the blank symbol, cost function)**.** Let Σ be an alphabet. Let λ be a symbol, called *blank symbol*, such that $\lambda \notin \Sigma$. We define $\Sigma_\lambda = \Sigma \cup \{\lambda\}$ and refer to the Σ_λ as the *alphabet with the blank symbol* λ. A *cost function* is a function $\gamma : \Sigma_\lambda \times \Sigma_\lambda \to \mathbb{R}$.

In the literature, it is often assumed that γ is a metric on Σ_λ [44, Page 202], [39, Definition 2.5], [45]; that is, γ satisfies for any $a, b, c \in \Sigma_\lambda$ the following conditions:

- (non-negativity, positivity) $\gamma(a, b) \geq 0$,
- (self-identity, separation) $\gamma(a, b) = 0$ if and only if $a = b$,
- (symmetry) $\gamma(a, b) = \gamma(b, a)$, and
- (triangle inequality) $\gamma(a, b) \leq \gamma(a, c) + \gamma(c, b)$.

We show an example of a cost function that satisfies the metric conditions in Table 2.1.

In this dissertation thesis, we often use a particular definition of a cost function where the costs for any pair of distinct symbols are set to 1, and the costs for two identical symbols are set to 0. We distinguish between this cost function and others by using the terms *unit* cost function and *non-unit* cost function.

▶ **Definition 2.22** (Unit cost function, non-unit cost function)**.** Let Σ_λ be an alphabet with the blank symbol λ equipped with a cost function γ. If for every $a, b \in \Sigma_\lambda$ it holds that

$$\gamma(a, b) = \begin{cases} 0 & \text{if } a = b, \\ 1 & \text{if } a \neq b, \end{cases}$$

then γ is called the *unit cost function*. Otherwise, it is called a *non-unit cost function*.

Note that the unit cost function satisfies the metric conditions. In the literature, the unit cost function is also known as the *discrete metric* or the *trivial metric* [44, Page 21].

Let γ be a cost function defined on Σ_λ. We use it to define the cost of 1-degree edit operations. Given $(a, b) \in \Sigma_\lambda \times \Sigma_\lambda$, it holds that if $a \neq b \neq \lambda$, then $\gamma(a, b)$ corresponds to the cost of relabeling operation that changes the label of a node from a to b. If $a = \lambda$ and $b \neq \lambda$, then $\gamma(a, b)$ corresponds to the cost of the insertion operation that inserts a new leaf labeled by b. Finally, if $a \neq \lambda$ and $b = \lambda$, then $\gamma(a, b)$ denotes the cost of the deletion operation that deletes a leaf labeled by a. Note that the pairs $(a, a) \in \Sigma_\lambda \times \Sigma_\lambda$ do not correspond to any operation and thus do not play a role in defining the cost of 1-degree edit operations.

▶ **Definition 2.23** (Cost of 1-degree edit operations)**.** Let Σ_λ be an alphabet with the blank symbol λ equipped with a cost function γ. Let $\mathcal{T} = (V, E, v_r, \preceq_{\mathrm{S}}, \mathsf{label})$ be an ordered labeled tree over Σ. Given $u, v \in V$ and $a \in \Sigma$, we define the *cost* of 1-degree edit operations as follows:

- the cost of operation $\mathsf{del}(v)$ is equal to $\gamma(\mathsf{label}(v), \lambda)$;
- the cost of operation $\mathsf{ins}(a, i, u)$, where $i \in \{1, 2, \ldots, \mathsf{degree}(u) + 1\}$ is equal to $\gamma(\lambda, a)$; and
- the cost of operation $\mathsf{rel}(v, a)$ is equal to $\gamma(\mathsf{label}(v), a)$.

We say that the 1-degree edit operations *have the unit cost* if γ is the unit cost function. Otherwise, we say that the 1-degree edit operations *have a non-unit cost*.

Using the cost of 1-degree edit operations, we define the *cost of an edit script* as the sum of the costs of operations in the script. An *optimal edit script* between two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ is an edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ with the minimal cost and this cost is called the 1-degree edit distance between $\mathcal{T}_1$ and $\mathcal{T}_2$.

▶ **Definition 2.24** (1-degree edit distance)**.** Let Σ_λ be an alphabet with the blank symbol λ equipped with a cost function. Given two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ over Σ, the *1-degree edit distance* is a function $d : Tr(\Sigma) \times Tr(\Sigma) \to \mathbb{R}$ such that $d(\mathcal{T}_1, \mathcal{T}_2)$ is the cost of an optimal edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$.

Since we use a cost function γ defined on Σ_λ to define the cost of 1-degree edit operations, it holds that if γ is a metric function, then the set of all ordered labeled trees with the 1-degree edit distance also forms a metric space.

Given two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$, seeking the cost of an optimal edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ in the case where 1-degree edit operations have the unit cost correspond to seeking the minimum number of 1-degree edit operations required to make $\mathcal{T}_1$ isomorphic to $\mathcal{T}_2$. In other words, the goal is to find the length of a *shortest edit script* between $\mathcal{T}_1$ and $\mathcal{T}_2$. We refer to this variant of the 1-degree edit distance as the *simple 1-degree edit distance*.

▶ **Definition 2.25** (Simple 1-degree edit distance)**.** Let Σ be an alphabet. Given two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ over Σ, the *simple 1-degree edit distance* is a function $d_\mathrm{s} : Tr(\Sigma) \times Tr(\Sigma) \to \mathbb{N}_0$ such that $d_\mathrm{s}(\mathcal{T}_1, \mathcal{T}_2)$ is the length of a shortest edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$.

▶ **Example 2.26** (1-degree edit distance, simple 1-degree edit distance)**.** Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the ordered labeled trees illustrated in Figure 2.3. We need at least five operations in order to transform $\mathcal{T}_1$ into $\mathcal{T}_2$. Thus, the edit script illustrated in Figure 2.3 is a shortest one which makes $d_\mathrm{s}(\mathcal{T}_1, \mathcal{T}_2) = 5$. Assuming that the 1-degree edit operations come with a cost as described by the cost function illustrated in Table 2.1, the cost of the edit script illustrated in Figure 2.3 is 40. That does not make the edit script optimal since we can also transform $\mathcal{T}_1$ into $\mathcal{T}_2$ via an edit script illustrated in Figure 2.4 which cost is 36. We easily check that we cannot do better which makes $d(\mathcal{T}_1, T_2) = 36$.

2.3.1.3 Tree substitution

In this section, we define four different kinds of tree substitution in an ordered labeled tree:

- Substitution of all occurrences of leaves labeled by the same symbol for possibly distinct trees, where every occurrence is replaced by a tree that may be different from trees used as replacements for other occurrences.
- Substitution of all occurrences of internal nodes labeled by the same symbol for possibly distinct linear trees or *empty trees*, where every occurrence is replaced either by the empty tree or by a linear tree that may be different from trees used as replacements for other occurrences.
- Concurrent substitution at leaves, where the same tree replaces all occurrences of leaves labeled by the same symbol.
- Concurrent substitution at internal nodes, where all occurrences of internal nodes labeled by the same symbol are replaced by the same linear tree or the empty tree.

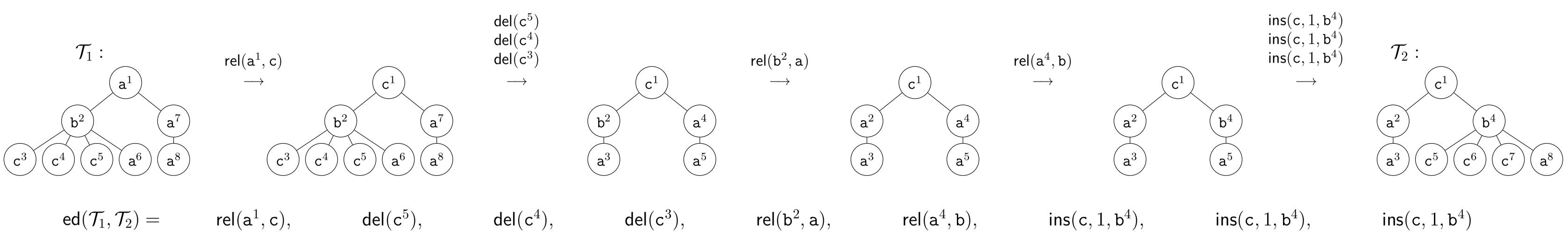


Figure 2.4 An example of two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ over alphabet $\{a, b, c\}$ with an edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$. Assuming that 1-degree edit operations are assigned costs according to the cost function γ described in Table 2.1, the cost of the edit script is 36 which makes it optimal.

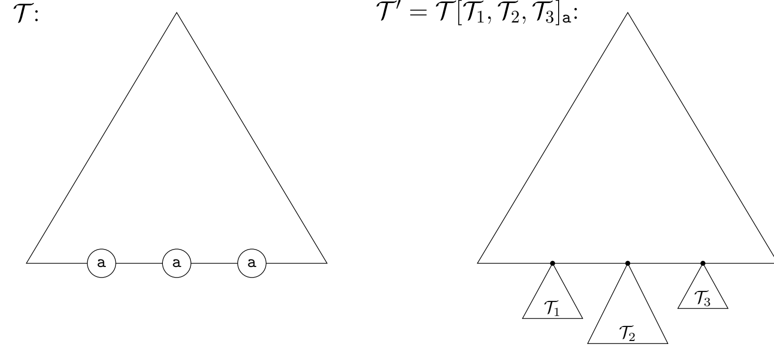


Figure 2.5 Illustration of substitution of all occurrences of leaves labeled by symbol a in an ordered labeled tree $\mathcal{T}$ for possibly distinct ordered labeled trees $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}_3$.

We note that our illustrative examples of the substitutions and the notation we introduce in the definitions are inspired by Cleophas [22, Section 3.1.1.1].

We start with the simplest substitution, which is the substitution of all occurrences of leaves labeled by the same symbol for possibly distinct trees. See Definition 2.27 and the illustration in Figure 2.5. We refer to this kind of tree substitution in Section 3.1.2, where we use it to specify when a pattern tree with *subtree wildcards* matches another tree.

▶ **Definition 2.27** (Substitution of all occurrences of leaves labeled by the same symbol for possibly distinct trees)**.** Let Σ be an alphabet. Let a be a symbol of Σ. Let $\mathcal{T} = (V, E, r, \preceq_{\mathrm{S}}, \mathsf{label})$ be an ordered labeled tree over Σ, where precisely $k \geq 0$ leaves is labeled by a. Let $l_1, \dots, l_k$ be those leaves in preorder. Let $(\mathcal{T}_1, \dots, \mathcal{T}_k)$ be a sequence of ordered labeled trees over Σ such that $\mathcal{T}_i = (V_i, E_i, r_i, \preceq_{\mathrm{S}_i}, \mathsf{label}_i)$. The tree substitution in $\mathcal{T}$ of the occurrences of a for $(\mathcal{T}_1, \dots, \mathcal{T}_k)$, denoted by $\mathcal{T}[\mathcal{T}_1, \dots, \mathcal{T}_k]_a$, is an ordered labeled tree $\mathcal{T}'$ defined as follows:

- If $|V| = 1$ and $k = 1$, then $\mathcal{T}' = \mathcal{T}_1$.
- Otherwise, $\mathcal{T}' = (V', E', r', \preceq'_{\mathrm{S}}, \mathsf{label}')$, where
  - $V' = V \setminus \{l_1, \dots, l_k\} \cup V_1 \cup \dots \cup V_k$,
  - $E' = E \setminus \{\{l_1, \mathsf{parent}(l_1)\}, \dots, \{l_k, \mathsf{parent}(l_k)\}\} \cup E_1 \cup \dots \cup E_k \cup \{r_1, \mathsf{parent}(l_1)\} \cup \dots \cup \{r_k, \mathsf{parent}(l_k)\}$,
  - $r' = r$,
  - $\mathsf{label}' = \mathsf{label} \setminus \{(l_1, a), \dots, (l_k, a)\} \cup \mathsf{label}_1 \cup \dots \cup \mathsf{label}_k$, and
  - for every pair of distinct nodes $u, v \in V'$, it holds that $u \prec'_{\mathrm{S}} v$ if and only if one of the following conditions is satisfied:
    - $u \prec_{\mathrm{S}} v$,
    - $u \prec_{\mathrm{S}_i} v$ for some $i \in \{1, \dots, k\}$,
    - $u = r_i$ and $l_i \prec_{\mathrm{S}} v$, or
    - $v = r_i$ and $u \prec_{\mathrm{S}} l_i$.

Next, we define the substitution of all occurrences of a symbol in the internal nodes. For simplicity, we assume that the internal nodes that are being substituted are not in the parent-child relationship. An internal node can be substituted for a linear ordered labeled tree. Additionally, we allow substitution for the *empty tree*, which we define as a graph where the vertex set and the edge set are empty. Intuitively, the substitution of an internal node v in a tree $\mathcal{T}$ for the empty tree corresponds to deleting v from $\mathcal{T}$ which makes the children of v to become children of

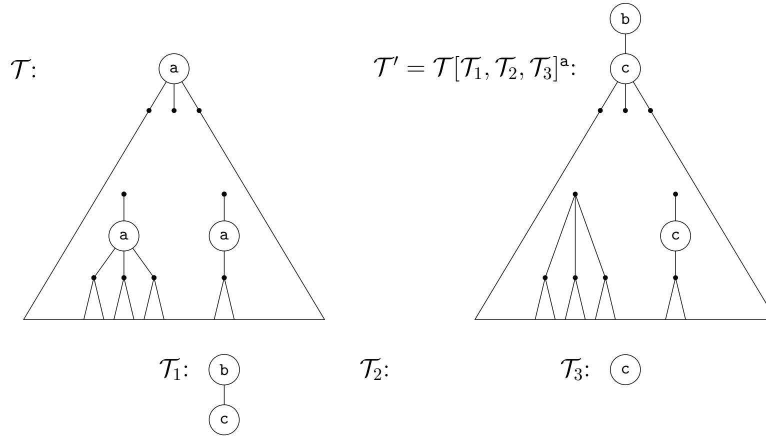


Figure 2.6 Illustration of substitution of all occurrences of internal nodes labeled by symbol $\mathtt{a}$ in an ordered labeled tree $\mathcal{T}$ for trees $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}_3$. Note that $\mathcal{T}_2$ is the empty tree.

$\mathsf{parent}(v)$. Since the root of a tree is also an internal node, it can also be substituted. However, if the root is to be substituted for the empty tree, we assume it has only one child, which becomes the new root.

▶ **Definition 2.28** (Substitution of all occurrences of a symbol in internal nodes by possibly distinct linear trees or empty trees)**.** Let Σ be an alphabet. Let a be a symbol of Σ. Let $\mathcal{T} = (V, E, r, \preceq_{\mathrm{S}}, \mathsf{label})$ be an ordered labeled tree over Σ, where precisely $k \geq 0$ internal nodes is labeled by a. Let $w_1, \ldots, w_k$ be those nodes in preorder. Assume that there are no nodes w_i, w_j for $i, j \in \{1, \ldots, k\}$ such that $\mathsf{parent}(w_i) = w_j$. Let $(\mathcal{T}_1, \ldots, \mathcal{T}_k)$ be a sequence, where $\mathcal{T}_i$ for each $i \in \{1, \ldots, k\}$ is either a linear ordered labeled tree $(V_i, E_i, r_i, \preceq_{\mathrm{S}_i}, \mathsf{label}_i)$ over Σ or the empty tree. Assume that if $w_i = r$ and $\mathcal{T}_i$ is the empty tree for some $i \in \{1, \ldots, k\}$, then $\mathsf{degree}(w_i) = 1$. The tree substitution in $\mathcal{T}$ of the occurrences of a for $(\mathcal{T}_1, \ldots, \mathcal{T}_k)$, denoted by $\mathcal{T}[\mathcal{T}_1, \ldots, \mathcal{T}_k]^a$, is an ordered labeled tree $\mathcal{T}'$ defined by:

- If $|V| = 1$, then $\mathcal{T}' = \mathcal{T}$.
- Otherwise, $\mathcal{T}' = (V', E', r', \preceq'_{\mathrm{S}}, \mathsf{label}')$ defined as follows:
  - $V' = V \setminus \{w_1, \ldots, w_k\} \cup V_1 \cup \cdots \cup V_k$.
  - $E' = E \setminus \{\{w_i, u\} : i \in \{1, \ldots, k\} \wedge u \in V\} \cup E_1 \cup \cdots \cup E_k \cup X_1 \cup \cdots \cup X_k$, where

$$X_i = \begin{cases} \{\{\mathsf{parent}(w_i), u\} : u \in \mathsf{children}(w_i)\} & \text{if } w_i \neq r \wedge V(\mathcal{T}_i) = \emptyset, \\ \{\mathsf{parent}(w_i), r_i\} \cup \{\{\mathsf{leaf}(\mathcal{T}_i), u\} : u \in \mathsf{children}(w_i)\} & \text{if } w_i \neq r \wedge V(\mathcal{T}_i) \neq \emptyset, \\ \emptyset & \text{if } w_i = r \wedge V(\mathcal{T}_i) = \emptyset, \\ \{\{\mathsf{leaf}(\mathcal{T}_i), u\} : u \in \mathsf{children}(w_i)\} & \text{if } w_i = r \wedge V(\mathcal{T}_i) \neq \emptyset. \end{cases}$$

  - $r' = \begin{cases} r & \text{if } w_i \neq r \text{ for every } i \in \{1, \ldots, k\}, \\ \mathsf{child}(w_i) & \text{if } w_i = r \wedge V(\mathcal{T}_i) = \emptyset, \\ r_i & \text{if } w_i = r \wedge V(\mathcal{T}_i) \neq \emptyset. \end{cases}$
  - $\mathsf{label}' = \mathsf{label} \setminus \{(w_1, a), \ldots, (w_k, a)\} \cup \{\mathsf{label}_i : i \in \{1, \ldots, k\} \wedge V(\mathcal{T}_i) \neq \emptyset\}$.
  - For every pair of distinct nodes $u, v \in V'$, it holds that $u \prec'_{\mathrm{S}} v$ if and only if one of the following conditions is satisfied:
    * $u \prec_{\mathrm{S}} v$,

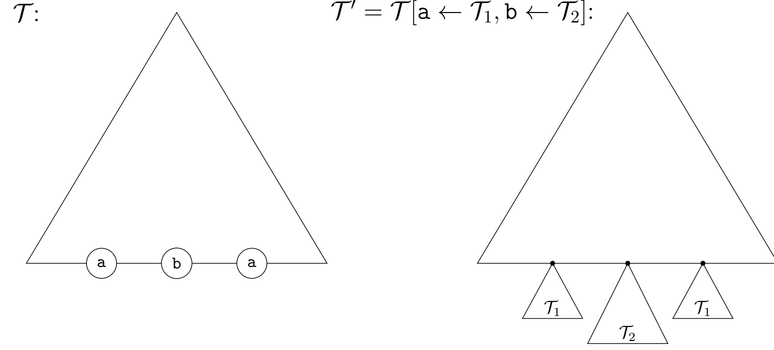


Figure 2.7 Concurrent substitution at leaves labeled by symbols $\mathtt{a}$ and $\mathtt{b}$ in an ordered labeled tree $\mathcal{T}$ for trees $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively.

- $u = r_i$ and $w_i \prec_S v$ for some $i \in \{1, \dots, k\}$, or
- $v = r_i$ and $u \prec_S w_i$ for some $i \in \{1, \dots, k\}$.

We illustrate the substitution of all occurrences of a symbol in internal nodes in Figure 2.6. We refer to this kind of tree substitution in Section 3.1.2, where we use it to specify when a pattern tree with *path wildcards* matches another tree.

We can now define two substitutions called concurrent. To define the concurrent substitution at leaves, where the same tree replaces all occurrences of leaves labeled by the same symbol, we refer to the tree substitution introduced by Definition 2.27. See the illustrative example in Figure 2.7. Then, we use the tree substitution introduced by Definition 2.28 to define the concurrent substitution at internal nodes—all occurrences of internal nodes labeled by the same symbol are replaced by the same linear tree or the empty tree. See the example in Figure 2.8. We use concurrent substitutions in Section 3.1.2 to specify when a pattern with *subtree variables* or *path variables* matches another tree.

▶ **Definition 2.29** (Concurrent substitution at leaves). Let Σ_1 and Σ_2 be disjoint alphabets. Let $\Sigma_2 = \{a_1, \dots, a_m\}$. Let $\mathcal{T}$ be an ordered labeled tree over $\Sigma_1 \cup \Sigma_2$. Assume that all nodes labeled by symbols from Σ_2 (if any) are leaves. Let $|\mathsf{occ}(a_i)|$ for $i \in \{1, \dots, m\}$ denote the number of leaves in $\mathcal{T}$ labeled by a_i. Let $(\mathcal{T}_1, \dots, \mathcal{T}_m)$ be a sequence of ordered labeled trees over Σ_1. The tree substitution of $a_1, \dots, a_m$ for $\mathcal{T}_1, \dots, \mathcal{T}_m$ in $\mathcal{T}$, denoted $\mathcal{T}[a_1 \leftarrow \mathcal{T}_1, \dots, a_m \leftarrow \mathcal{T}_m]$, is an ordered labeled tree $\mathcal{T}'$ defined as follows:

- If $m = 1$, then $\mathcal{T}' = \mathcal{T}[\underbrace{\mathcal{T}_1, \dots, \mathcal{T}_1}_{|\mathsf{occ}(a_1)|}]_{a_1}$.
- If $m \geq 2$, then $\mathcal{T}' = \hat{\mathcal{T}}[a_2 \leftarrow \mathcal{T}_2, \dots, a_m \leftarrow \mathcal{T}_m]$, where $\hat{\mathcal{T}} = \mathcal{T}[\underbrace{\mathcal{T}_1, \dots, \mathcal{T}_1}_{|\mathsf{occ}(a_1)|}]_{a_1}$.

▶ **Definition 2.30** (Concurrent substitution at internal nodes). Let Σ_1 and Σ_2 be disjoint alphabets. Let $\Sigma_2 = \{a_1, \dots, a_m\}$. Let $\mathcal{T}$ be an ordered labeled tree over $\Sigma_1 \cup \Sigma_2$. Assume that all nodes labeled by symbols from Σ_2 (if any) are internal nodes. Assume that there are no nodes $u, v \in \mathcal{T}$ labeled by the same symbol of Σ_2 such that $\mathsf{parent}(u) = v$. Let $|\mathsf{occ}(a_i)|$ for $i \in \{1, \dots, m\}$ denote the number of nodes in $\mathcal{T}$ labeled by a_i. Let $(\mathcal{T}_1, \dots, \mathcal{T}_m)$ be a sequence, where $\mathcal{T}_i$ for each $i \in \{1, \dots, m\}$ is either a linear ordered labeled tree over Σ_1 or the empty tree. Assume that if the root of $\mathcal{T}$ is labeled by a_i and $\mathcal{T}_i$ is the empty tree for some $i \in \{1, \dots, m\}$, then $\mathsf{degree}(\mathsf{root}(\mathcal{T})) = 1$. The tree substitution of $a_1, \dots, a_m$ for $\mathcal{T}_1, \dots, \mathcal{T}_m$ in $\mathcal{T}$, denoted $\mathcal{T}[a_1 \leftarrow \mathcal{T}_1, \dots, a_m \leftarrow \mathcal{T}_m]$, is an ordered labeled tree $\mathcal{T}'$ defined as follows:

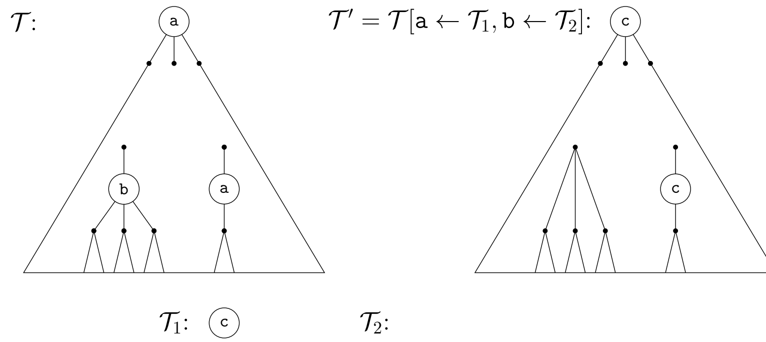


Figure 2.8 Concurrent substitution at internal nodes labeled by symbols $\mathtt{a}$ and $\mathtt{b}$ in an ordered labeled tree $\mathcal{T}$ for trees $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. Note that $\mathcal{T}_2$ is the empty tree.

- If $m = 1$, then $\mathcal{T}' = \mathcal{T}[\underbrace{\mathcal{T}_1, \ldots, \mathcal{T}_1}_{|\,\mathsf{occ}(a_1)|}]^{a_1}$.
- If $m \geq 2$, then $\mathcal{T}' = \hat{\mathcal{T}}[a_2 \leftarrow \mathcal{T}_2, \ldots, a_m \leftarrow \mathcal{T}_m]$, where $\hat{\mathcal{T}} = \mathcal{T}[\underbrace{\mathcal{T}_1, \ldots, \mathcal{T}_1}_{|\,\mathsf{occ}(a_1)|}]^{a_1}$.

2.3.2 Tree parts

This section contains definitions of several tree parts used in tree pattern matching. Moreover, we discuss similarities between tree parts and string parts defined in Section 2.1.2.

Given an ordered labeled tree, we first define its subtree, bottom-up subtree, and top-down subtree. Our definitions agree with the ones given by Valiente [39]. Then, we define notions of stringpath and rootpath. This terminology has been used, for example, by Cleophas [22]. In addition, we define a path of an ordered labeled tree as a special case of subtree.

▶ **Definition 2.31** (Subtree)**.** Let $\mathcal{T} = (V, E, r, \preceq_{\mathrm{S}}, \mathsf{label})$ be an ordered labeled tree with imposed vertical partial ordering $\preceq_{\mathrm{V}}$ on V. A *subtree* of $\mathcal{T}$ is an ordered labeled tree $\mathcal{T}' = (V', E', r', \preceq'_{\mathrm{S}}, \mathsf{label}')$, where

- $V' \subseteq V$,
- $E' \subseteq E$,
- $r' \in V'$ such that there is no node $v \in V'$ for which $v \prec_{\mathrm{V}} r'$,
- $\preceq'_{\mathrm{S}}$ is the restriction of $\preceq_{\mathrm{S}}$ to V', and
- label' is the restriction of label to V.

A subtree is a concept similar to that of a substring because in both cases, we are interested in a connected part of a given object: substrings are contiguous sequences of symbols, and subtrees are connected subgraphs. Furthermore, if we represent a string $\boldsymbol{x}$ as a linear ordered labeled tree $\mathcal{T}$, these two notions can be used interchangeably: a substring of $\boldsymbol{x}$ corresponds to a subtree of $\mathcal{T}$ and vice versa. See an illustration in Figure 2.9a.

A particular case of a subtree is a *bottom-up subtree*. If a node is part of a bottom-up subtree, so are all its descendants.

▶ **Definition 2.32** (Bottom-up subtree)**.** Let $\mathcal{T}$ be an ordered labeled tree and $\mathcal{S}$ be its subtree. If $\mathsf{children}(v) \subseteq V(\mathcal{S})$ for every node $v \in V(\mathcal{S})$, then S is called a *bottom-up subtree* of $\mathcal{T}$.

Given an ordered labeled tree $\mathcal{T}$, a bottom-up subtree of $\mathcal{T}$ can be uniquely described by choosing a node $v \in V(\mathcal{T})$ to be its root. We denote a bottom-up subtree of $\mathcal{T}$ with the root $v \in V(\mathcal{T})$ by $\mathcal{T}/v$. A bottom-up subtree $\mathcal{T}/v$ is called *proper* if $v \neq \mathsf{root}(\mathcal{T})$.

A bottom-up subtree is similar to a suffix since we are interested in a connected part at the end of a given object in both cases. Furthermore, if we again represent a string $\boldsymbol{x}$ as a linear ordered labeled tree $\mathcal{T}$, every suffix of $\boldsymbol{x}$ corresponds to a bottom-up subtree of $\mathcal{T}$ and vice versa. Moreover, just as every suffix is a substring, it holds that every bottom-up subtree is a subtree. See an illustration in Figure 2.9b.

Another particular case of a subtree is a *top-down subtree* for which it holds that if a node is in its node set, so is its parent. Therefore, the root of a top-down subtree of a tree $\mathcal{T}$ is always the root of $\mathcal{T}$.

▶ **Definition 2.33** (Top-down subtree). Let $\mathcal{T}$ be an ordered labeled tree and $\mathcal{S}$ be its subtree. If $\mathsf{parent}(v) \in V(\mathcal{S})$ for every node $v \in V(\mathcal{S}) \setminus \{\mathsf{root}(\mathcal{S})\}$, then $\mathcal{S}$ is called a *top-down subtree* of $\mathcal{T}$.

A top-down subtree of a tree is similar to a prefix of a string in the following way: a prefix is a contiguous sequence of symbols that starts at the beginning of a string; similarly, a top-down subtree is a connected part of a tree that starts at the beginning of the tree, that is, with its root. Moreover, every prefix is a substring, just as every top-down subtree is a subtree. Also, for a string $\boldsymbol{x}$ represented as a linear ordered labeled tree $\mathcal{T}$, we can use both notions interchangeably—every prefix of $\boldsymbol{x}$ is a top-down subtree of $\mathcal{T}$ and vice versa. See an illustration in Figure 2.9c.

Apart from a subtree, a bottom-up subtree, and a top-down subtree, we also define three other parts of a tree which can be seen as paths in its underlying graph. Specifically, we define a *path*, a *rootpath*, and a *stringpath*.

▶ **Definition 2.34** (Path). Let $\mathcal{T}$ be an ordered labeled tree. A subtree of $\mathcal{T}$ is said to be a *path* of $\mathcal{T}$ if it is a linear tree.

▶ **Definition 2.35** (Rootpath). Let $\mathcal{T}$ be an ordered labeled tree. A path of $\mathcal{T}$ is said to be a *rootpath* of $\mathcal{T}$ if its root is the root of $\mathcal{T}$.

A rootpath of an ordered labeled tree $\mathcal{T}$ can be uniquely described by choosing a node $v \in V(\mathcal{T})$ to be the leaf of the rootpath. We denote the rootpath of $\mathcal{T}$ with the leaf v by $\mathcal{T}|_v$. To uniquely describe a path of a tree, we need two nodes since its root can be different from the root of the tree. We denote a path of $\mathcal{T}$ with the root u and the leaf v by $\mathcal{T}|_v^u$.

▶ **Definition 2.36** (Stringpath). Let $\mathcal{T}$ be an ordered labeled tree. A rootpath $\mathcal{T}|_v$, where $v \in V(\mathcal{T})$, is said to be a *stringpath* of $\mathcal{T}$ if v is a leaf of $\mathcal{T}$.

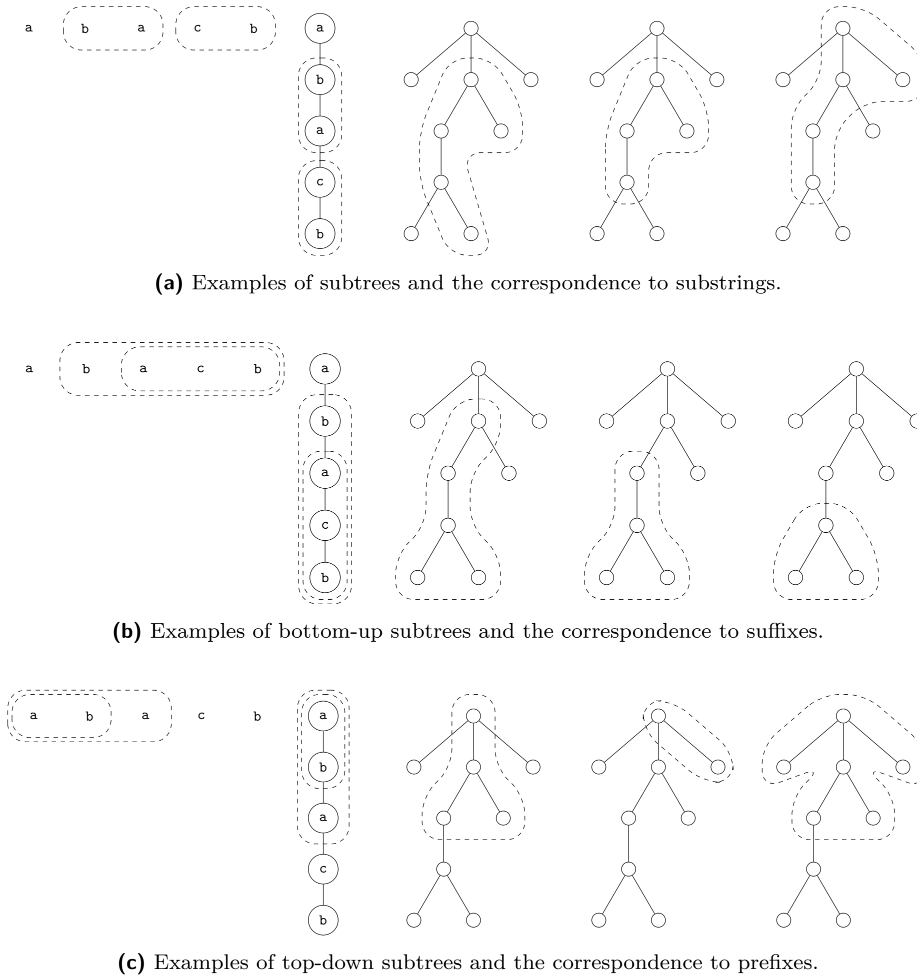
2.4 Tree languages

In this section, we extend the notion of a (string) language to trees. We also show that tree languages can be represented as string languages by encoding trees as strings using a linear tree notation. Specifically, we show how to unambiguously represent ordered labeled trees as strings using prefix bar notation and path notation. Moreover, we briefly discuss XML as another way of tree representation and its query language XPath.

In general, just as string languages are defined as sets of strings, *tree languages* are defined as sets of trees. Since our focus is on ordered labeled trees, we define the tree language as a set of ordered labeled trees.

▶ **Definition 2.37** (Tree language). Let Σ be an alphabet. A *tree language* over Σ is a subset of $Tr(\Sigma)$.

In this dissertation thesis, we propose methods that process trees using (string) automata. As a result, we need to encode trees as strings. A function that for a given tree returns its



(a) Examples of subtrees and the correspondence to substrings.

(b) Examples of bottom-up subtrees and the correspondence to suffixes.

(c) Examples of top-down subtrees and the correspondence to prefixes.

Figure 2.9 Examples of subtrees, bottom-up subtrees, and top-down subtrees of an ordered labeled tree and the correspondence to string parts.

string representation is called a *linear tree notation*. A well-known linear tree notation for ordered labeled trees is the *nested parenthesis notation* in which an ordered labeled tree over an alphabet Σ is represented as a string over alphabet $\Sigma \cup \{(,)\}$. The notation is based on the preorder tree traversal, where every internal node is described by its label followed by an ordered list of its children. For example, the parenthesis notation of tree $\mathcal{T}_2$ in Figure 2.2 is $\mathtt{a}\,(\mathtt{b}\,(\mathtt{a}(), \mathtt{b}(\mathtt{a}()))\,, \mathtt{a}\,(\mathtt{b}()))$.

In this dissertation thesis, we use two linear notations for ordered labeled trees, called the prefix bar notation and the path notation. We describe both of these notations in the following sections.

2.4.1 Prefix bar notation for ordered labeled trees

Stoklasa, Janoušek, and Melichar introduced the *prefix bar notation* at London Stringology Days in 2010 [66]; see also Janoušek [23, Chapter 5] or Flouri [30, Chapter 3]. The notation is a simplified variant of the nested parentheses notation. The simplification is based on the observation that the left parenthesis is redundant since there is always the root of a bottom-up

subtree immediately preceding it. The prefix bar notation also does not use commas since, even without them, there is only one way to parse such strings as ordered labeled trees.

▶ **Definition 2.38** (Prefix bar notation for ordered labeled trees [30]). Let Σ be an alphabet such that $\uparrow \notin \Sigma$. The *prefix bar notation* is a function $\mathsf{prefBar} : Tr(\Sigma) \to (\Sigma \cup \{\uparrow\})^+$ defined for an ordered labeled tree $\mathcal{T} = (V, E, r, \preceq_S, \mathsf{label})$ over Σ as follows:

$$\mathsf{prefBar}(\mathcal{T}) = \begin{cases} \mathsf{label}(r) \uparrow & \text{if } r \text{ is a leaf,} \\ \mathsf{label}(r)\, \mathsf{prefBar}(\mathcal{T}/v_1)\, \mathsf{prefBar}(\mathcal{T}/v_2) \dots \mathsf{prefBar}(\mathcal{T}/v_k) \uparrow & \text{otherwise,} \end{cases}$$

where $\{v_1, \dots, v_k\} = \mathsf{children}(r)$ such that $v_1 \prec_S v_2 \prec_S \cdots \prec_S v_k$.

In other words, the prefix bar notation of an ordered labeled tree lists the node labels using the preorder traversal of the tree where a special symbol called *bar*, denoted by "$\uparrow$", is used to encode hierarchical relationships between nodes in the tree—every bar symbol indicates the end of a bottom-up subtree. For example, the prefix bar notations of trees $\mathcal{T}_1$ and $\mathcal{T}_2$ illustrated in Figure 2.2 are $\mathtt{aac} \uparrow \mathtt{bc} \uparrow\uparrow\uparrow \mathtt{bc} \uparrow\uparrow\uparrow$ and $\mathtt{aba} \uparrow \mathtt{ba} \uparrow\uparrow\uparrow \mathtt{ab} \uparrow\uparrow\uparrow$, respectively. Note that the length of $\mathsf{prefBar}(\mathcal{T})$ is equal to $2|\mathcal{T}|$ for every ordered labeled tree $\mathcal{T}$.

▶ Convention 2.39. We assume in the further text that the bar symbol is not included in any alphabet. That is, $\uparrow \notin \Sigma$ for every alphabet Σ. Moreover, we abbreviate $\Sigma \cup \{\uparrow\}$ to $\Sigma_\uparrow$.

Let $\mathcal{T}$ be an ordered labeled tree. Every node $v \in V(\mathcal{T})$ is in $\mathsf{prefBar}(\mathcal{T})$ associated with two positions that we call *label position* and *bar position*. The label position corresponds to the position of the label of v in $\mathsf{prefBar}(\mathcal{T})$, and the bar position corresponds to the position of the bar symbol in $\mathsf{prefBar}(\mathcal{T})$ indicating the end of the bottom-up subtree $\mathcal{T}/v$. Thus, every node v can be associated with a pair (i, j), called a *label-bar pair*, where i is the label position and j is the bar position.

It is easy to see that two ordered labeled trees have the same prefix bar notation if and only if they are isomorphic. Thus, we can unambiguously transform any ordered labeled tree over Σ into a string over $\Sigma_\uparrow$ and back using the prefix bar notation. However, not every string over $\Sigma_\uparrow$ represents an ordered labeled tree. The correspondence between strings over $\Sigma_\uparrow$ and the prefix bar notation of ordered labeled trees over Σ is summarized by Definition 2.40 and Lemma 2.41.

▶ **Definition 2.40** (Bar checksum [30]). Let Σ be an alphabet. The *bar checksum* is a function $\mathsf{barCheck} : \Sigma_\uparrow^+ \to \mathbb{N}_0$ that is for a nonempty string $\boldsymbol{x}$ over $\Sigma_\uparrow$ defined as $\mathsf{barCheck}(\boldsymbol{x}) = \sum_{i=1}^{|\boldsymbol{x}|} \mathsf{bc}(\boldsymbol{x}_i)$, where

$$\mathsf{bc}(\boldsymbol{x}_i) = \begin{cases} 1 & \text{if } \boldsymbol{x}_i = \uparrow, \\ -1 & \text{if } \boldsymbol{x}_i \in \Sigma. \end{cases}$$

▶ **Lemma 2.41** (Correspondence between strings over $\Sigma_\uparrow$ and ordered labeled trees). *Let Σ be an alphabet. A nonempty string $\boldsymbol{x}$ over $\Sigma_\uparrow$ is the prefix bar notation of an ordered labeled tree over Σ if and only if $\mathsf{barCheck}(\boldsymbol{x}) = 0$ and $\mathsf{barCheck}(\boldsymbol{y}) < 0$ for every proper prefix $\boldsymbol{y}$ of $\boldsymbol{x}$.*

Proof. The proof is divided into two parts.

($\Longrightarrow$) Assume $\boldsymbol{x}$ is the prefix bar notation of an ordered labeled tree $\mathcal{T} = (V, E, r, \preceq_S, \mathsf{label})$. We show that $\mathsf{barCheck}(\boldsymbol{x}) = 0$ and $\mathsf{barCheck}(\boldsymbol{y}) < 0$ for every proper prefix $\boldsymbol{y}$ of $\boldsymbol{x}$ by induction on the depth of $\mathcal{T}$:

- Assume $\mathsf{depth}(\mathcal{T}) = 0$. Then, $\boldsymbol{x} = \mathsf{label}(r) \uparrow$. Thus, $\mathsf{barCheck}(\boldsymbol{x}) = -1 + 1 = 0$. Also, the only proper prefix of $\boldsymbol{x}$ is string $\boldsymbol{y} = \mathsf{label}(r)$ for which $\mathsf{barCheck}(\boldsymbol{y}) = -1 < 0$. Hence, the claim holds for tree with depth 0.

- Assume $\mathsf{depth}(\mathcal{T}) \geq 1$ and that the claim holds for trees with depths $0, \ldots, \mathsf{depth}(\mathcal{T})-1$. Then, $\boldsymbol{x} = \mathsf{label}(r)\,\mathsf{prefBar}(\mathcal{T}/v_1)\,\mathsf{prefBar}(\mathcal{T}/v_2)\ldots\mathsf{prefBar}(\mathcal{T}/v_k)\uparrow$, where $\{v_1, \ldots, v_k\} = \mathsf{children}(r)$ such that $v_1 \prec_S v_2 \prec_S \cdots \prec_S v_k$. Thus,

$$\mathsf{barCheck}(\boldsymbol{x}) = -1 + \sum_{i=1}^{k} \mathsf{barCheck}(\mathsf{prefBar}(\mathcal{T}/v_i)) + 1.$$

  By the induction hypothesis, we get that $\mathsf{barCheck}(\mathsf{prefBar}(\mathcal{T}/v_i)) = 0$ for each $i \in \{1, \ldots, k\}$. Thus, $\mathsf{barCheck}(\boldsymbol{x}) = 0$. Also, since $\mathsf{barCheck}(\boldsymbol{y}) < 0$ for every proper prefix of $\mathsf{prefBar}(\mathcal{T}/v_i)$ for each $i \in \{1, \ldots, k\}$, it follows that the bar checksum is also negative for every proper prefix of $\boldsymbol{x}$. Hence, the claim holds.

($\Longleftarrow$) Assume $\mathsf{barCheck}(\boldsymbol{x}) = 0$ and $\mathsf{barCheck}(\boldsymbol{y}) < 0$ for every proper prefix $\boldsymbol{y}$ of $\boldsymbol{x}$. Since each $\boldsymbol{x}_i$ either increases the checksum by 1 (for $\boldsymbol{x}_i = \uparrow$) or reduces it by 1 (for $\boldsymbol{x}_i \in \Sigma$), it follows that $\boldsymbol{x}$ contains the same number of bar symbols as symbols from Σ in order to $\mathsf{barCheck}(\boldsymbol{x}) = 0$. Thus, $|\boldsymbol{x}|$ is even. We show that $\boldsymbol{x}$ is the prefix bar notation of an ordered labeled tree by induction on the length of $\boldsymbol{x}$:

- Assume $|\boldsymbol{x}| = 2$. Then, the only possible form for $\boldsymbol{x}$ is $\boldsymbol{x} = a\uparrow$, where $a \in \Sigma$, which is the prefix bar notation of an ordered labeled tree. Hence, the claim holds for $|\boldsymbol{x}| = 2$.

- Assume $|\boldsymbol{x}| \geq 4$ and that the claim holds for strings with lengths $2, 4, 6, \ldots, |\boldsymbol{x}| - 2$. Since $\mathsf{barCheck}(\boldsymbol{y}) < 0$ for every proper prefix $\boldsymbol{y}$ of $\boldsymbol{x}$, we get that $\boldsymbol{x}_1 \in \Sigma$. Moreover, we get that $\boldsymbol{x}_{|\boldsymbol{x}|} = \uparrow$, because if $\mathsf{barCheck}(\boldsymbol{x}_1\boldsymbol{x}_2\ldots\boldsymbol{x}_{|\boldsymbol{x}|-1}) < 0$, the only possibility for $\mathsf{barCheck}(\boldsymbol{x}) = 0$ to hold is that $\mathsf{barCheck}(\boldsymbol{x}_1\boldsymbol{x}_2\ldots\boldsymbol{x}_{|\boldsymbol{x}|-1}) = -1$ and $\boldsymbol{x}_{|\boldsymbol{x}|} = \uparrow$. Let $\boldsymbol{w} = \boldsymbol{x}_2\boldsymbol{x}_3\ldots\boldsymbol{x}_{|\boldsymbol{x}|-1}$. Since $\mathsf{bc}(\boldsymbol{x}_1) = -1$ and $\mathsf{bc}(\boldsymbol{x}_{|\boldsymbol{x}|}) = 1$, we have that $\mathsf{barCheck}(\boldsymbol{w}) = 0$. Moreover, we get that $\mathsf{barCheck}(\boldsymbol{z}) \leq 0$ for every prefix $\boldsymbol{z}$ of $\boldsymbol{w}$. Our goal is to show that we can split $\boldsymbol{w}$ into $k \geq 1$ parts such that $\boldsymbol{w} = \boldsymbol{w}^1\boldsymbol{w}^2\ldots\boldsymbol{w}^k$, where $\mathsf{barCheck}(\boldsymbol{w}^i) = 0$ and $\mathsf{barCheck}(\boldsymbol{z}^i) < 0$ for every proper prefix $\boldsymbol{z}^i$ of $\boldsymbol{w}^i$ and every $i \in \{1, \ldots, k\}$. The splitting process is as follows: We obtain $\boldsymbol{w}^1$ by finding the first position $j \in \{1, \ldots, |\boldsymbol{w}|\}$ in $\boldsymbol{w}$ for which $\mathsf{barCheck}(\boldsymbol{w}_{1,\ldots,j}) = 0$. For the remaining nonempty string $\boldsymbol{w}_{j+1}\boldsymbol{w}_{j+2}\ldots\boldsymbol{w}_{|\boldsymbol{w}|}$ (if any), it again holds that its bar checksum is 0 and that its every prefix has the bar checksum less or equal to 0. Thus, we obtain $\boldsymbol{w}^2$ by again finding the first position such that the prefix ending at that position has the bar checksum 0. We continue in this fashion until the end of the string $\boldsymbol{w}$ is reached. By the induction hypothesis, we have that strings $\boldsymbol{w}^1, \ldots, \boldsymbol{w}^k$ are prefix bar notations of ordered labeled trees $\mathcal{T}_1, \ldots, \mathcal{T}_k$, respectively. Therefore, $\boldsymbol{x} = a\,\mathsf{prefBar}(\mathcal{T}_1)\,\mathsf{prefBar}(\mathcal{T}_2)\ldots\mathsf{prefBar}(\mathcal{T}_k)\uparrow$, where $a \in \Sigma$, which is the prefix bar notation of an ordered labeled tree. Hence, the claim holds. ◀

We use $L(\Sigma, \uparrow)$ to denote the (string) language over $\Sigma_\uparrow$ consisting of all strings that represent the prefix bar notation of ordered labeled trees over Σ. Since we can unambiguously convert every ordered labeled tree $\mathcal{T} \in Tr(\Sigma)$ to a string $\boldsymbol{x} \in L(\Sigma, \uparrow)$ and vice versa, every tree language can be represented as a string language using the prefix bar notation.

The prefix bar notation comes with a useful property that we call a *substring property*. The substring property is used in tree pattern matching methods that are part of arbology research. We also use this property in Chapter 5, where we propose automata-based methods for inexact tree pattern matching.

▶ **Lemma 2.42** (Substring property of the prefix bar notation [30]). *Let $\mathcal{T}$ be an ordered labeled tree and let $\mathcal{S}$ be its bottom-up subtree. Then $\mathsf{prefBar}(\mathcal{S})$ is a substring of $\mathsf{prefBar}(\mathcal{T})$.*

However, not every substring of $\mathsf{prefBar}(\mathcal{T})$ is a bottom-up subtree of tree $\mathcal{T}$. This is because $\mathcal{T}$ has $|\mathcal{T}|$ bottom-up subtrees (each node is the root of one bottom-up subree) but the maximum number of substrings occurring in a string (the prefix bar notation of $\mathcal{T}$) can be quadratic to

the length of the string. Only substrings which themselves are trees in the prefix bar notation represent bottom-up subtrees.

▶ **Lemma 2.43** (Correspondence between substrings of the prefix bar notation and bottom-up subtrees [30]). *Let $\mathcal{T}$ be an ordered labeled tree. A substring $\boldsymbol{x}$ of* $\mathsf{prefBar}(\mathcal{T})$ *represents a bottom-up subtree of $\mathcal{T}$ if and only if $\boldsymbol{x} \in L(\Sigma, \uparrow)$.*

The prefix bar notation can be used for both ranked and unranked trees. However, since labels in ranked trees determine the number of children of each node, we can simplify the notation by omitting the bar symbol. This simplified notation is called the *prefix notation* [30, Definition 52], [23, Section 3.3]. We note that the resulting strings are also known as *(ordered ranked) ground terms* in the literature [54, Page 109].

2.4.2 Path notation for ordered labeled trees

An ordered labeled tree can be described by a string representation that is based on its stringpath set. We call this notation the *path notation*. In the literature, it is also known as the *path language* [21, Page 43], [67], or referred to as *tree stringpaths* [22, Definition 3.1.17].

The path notation for an ordered labeled tree is a set of strings where each string is an encoded stringpath. Specifically, each stringpath is encoded in the form of $a_1 i_1 a_2 i_2 \dots a_{l-1} i_{l-1} a_l$, where l is the number of nodes in the stringpath, $a_1, \dots, a_l$ are its node labels, and i_j for each $j \in \{1, \dots, l-1\}$ is a number indicating that a_{j+1} is the label of the node that is i_j-th child of node represented by a_j.

▶ **Definition 2.44** (Path notation for ordered labeled trees). Let Σ be an alphabet such that $\Sigma \cap \mathbb{N} = \emptyset$. The *path notation* is a function $\mathsf{path} : Tr(\Sigma) \to 2^X$, where $X = (\Sigma \cdot \mathbb{N})^* \cdot \Sigma$, defined for an ordered labeled tree $\mathcal{T} = (V, E, r, \preceq_S, \mathsf{label})$ over Σ as follows:

$$\mathsf{path}(\mathcal{T}) = \begin{cases} \{\mathsf{label}(r)\} & \text{if } r \text{ is a leaf,} \\ \bigcup_{i=1}^{\mathsf{degree}(r)} \{\mathsf{label}(r) i \boldsymbol{w} : \boldsymbol{w} \in \mathsf{path}(\mathcal{T}/v_i)\} & \text{otherwise,} \end{cases}$$

where $\{v_1, \dots, v_{\mathsf{degree}(r)}\} = \mathsf{children}(r)$ such that $v_1 \prec_S v_2 \prec_S \cdots \prec_S v_{\mathsf{degree}(r)}$.

In other words, the path notation for an ordered labeled tree over Σ is a string language over $\Sigma \cup \{1, 2, \dots, k\}$ where numbers are used to encode the order of siblings and k denotes the maximum degree of nodes in the tree. For example, the path notation for trees $\mathcal{T}_1$ and $\mathcal{T}_2$ illustrated in Figure 2.2 is $\{\mathtt{a1a1c}, \mathtt{a1a2b1c}, \mathtt{a2b1c}\}$ and $\{\mathtt{a1b1a}, \mathtt{a1b2b1a}, \mathtt{a2a1b}\}$, respectively.

▶ Convention 2.45. To avoid confusion, we assume that no alphabet contains numbers as symbols. That is, sets Σ and $\mathbb{N}$ are disjoint for every alphabet Σ. Moreover, we abbreviate $\Sigma \cup \{1, 2, \dots, k\}$ to $\Sigma_{\leq k}$.

Clearly, path notation is an unambiguous representation of an ordered labeled tree. Two ordered labeled trees have the same path notation if and only if they are isomorphic. Thus, we can transform any ordered labeled tree over Σ where k denotes the maximum degree of nodes into a string language over $\Sigma_{\leq k}$ and back using the path notation. However, not every string language over $\Sigma_{\leq k}$ represents an ordered labeled tree. We examine the correspondence between ordered labeled trees and string languages over $\Sigma_{\leq k}$ in Lemma 2.46. Moreover, we note that the numbers are necessary for the path notation to be an unambiguous representation of a tree. For example, consider a tree whose path notation is $\{\mathtt{a1a1b}, \mathtt{a1a2c}\}$. If we omit the numbers, we get $\{\mathtt{aab}, \mathtt{aac}\}$. However, this set represents two non-isomorphic trees: the tree $\{\mathtt{a1a1b}, \mathtt{a1a2c}\}$ and the tree $\{\mathtt{a1a1b}, \mathtt{a2a1c}\}$.

▶ **Lemma 2.46** (Correspondence between string languages over $\Sigma_{\leq k}$ and ordered labeled trees). *Let Σ be an alphabet. Let $k \geq 1$. A nonempty string language L over $\Sigma_{\leq k}$ is the path notation of an ordered labeled tree over Σ if and only if all of the following five conditions hold:*

1. *(root)* $\boldsymbol{x}_1 = \boldsymbol{y}_1$ *for every pair of strings* $\boldsymbol{x}, \boldsymbol{y} \in L$*;*

2. *(form) for every* $\boldsymbol{x} \in L$*, we have that* $|\boldsymbol{x}| = 2n+1$ *for some* $n \in \mathbb{N}_0$ *and for each* $i \in \{1, \dots, |\boldsymbol{x}|\}$*, we have that*
$$\boldsymbol{x}_i = \begin{cases} a \in \Sigma & \text{if } i \text{ is odd,} \\ i \in \{1, \dots, k\} & \text{if } i \text{ is even.} \end{cases}$$

3. *(sibling) for every* $\boldsymbol{x} \in L$ *and every* $i \in \{2, 4, 6 \dots, |\boldsymbol{x}| - 1\}$*, we have that if* $\boldsymbol{x}_i = c$*, where* $c \in \{2, 3, \dots, k\}$*, then there exists* $\boldsymbol{y} \in L$ *such that* $y_{1 \dots (i-1)} = x_{1 \dots (i-1)}$ *and* $\boldsymbol{y}_i = c - 1$*;*

4. *(integrity) for every pair of strings* $\boldsymbol{x}, \boldsymbol{y} \in L$ *and every position* $i \in \{2, 4, 6 \dots, \min(|\boldsymbol{x}|, |\boldsymbol{y}|) - 1\}$*, we have that if* $\boldsymbol{x}_{1 \dots i} = \boldsymbol{y}_{1 \dots i}$ *and* $\boldsymbol{x}_i, \boldsymbol{y}_i \in \{1, \dots, k\}$*, then* $\boldsymbol{x}_{i+1} = \boldsymbol{y}_{i+1}$*; and*

5. *(prefix) for every pair of distinct strings* $\boldsymbol{x}, \boldsymbol{y} \in L$ *such that* $|\boldsymbol{x}| < |\boldsymbol{y}|$*, we have that* $\boldsymbol{x}$ *is not prefix of* $\boldsymbol{y}$*.*

Proof. The proof is divided into two parts.

$(\Longrightarrow)$ Assume L is the path notation of an ordered labeled tree $\mathcal{T} = (V, E, r, \preceq_S, \mathsf{label})$. We show that all five conditions hold for L by induction on the depth of $\mathcal{T}$:

- Assume $\mathsf{depth}(\mathcal{T}) = 0$. Then, $L = \{\mathsf{label}(r)\}$. It is easy to check that all five conditions hold for L. Thus, the claim holds for tree with depth 0.

- Assume $\mathsf{depth}(\mathcal{T}) \geq 1$ and that the claim holds for trees with depths $0, \dots, \mathsf{depth}(\mathcal{T}) - 1$. Then,
$$L = \bigcup_{i=1}^{\mathsf{degree}(r)} \{\mathsf{label}(r) i \boldsymbol{w} : \boldsymbol{w} \in \mathsf{path}(\mathcal{T}/v_i)\},$$
where $\{v_1, \dots, v_{\mathsf{degree}(r)}\} = \mathsf{children}(r)$ such that $v_1 \prec_S v_2 \prec_S \cdots \prec_S v_{\mathsf{degree}(r)}$. By the induction hypothesis, we have that all conditions hold for $\mathsf{path}(\mathcal{T}/v_i)$ for every $i \in \{1, \dots, \mathsf{degree}(r)\}$. It is easy to check that by adding $\mathsf{label}(r)i$ as the prefix to strings $\boldsymbol{w} \in \mathsf{path}(\mathcal{T}/v_i)$ for every $i \in \{1, \dots, \mathsf{degree}(r)\}$ the conditions still hold. Hence, the claim holds.

$(\Longleftarrow)$ Assume that all five conditions hold for L. We show that L is the path notation of an ordered labeled tree by induction on the size of L:

- Assume $|L| = 1$. We show that L is the path notation of an ordered labeled tree by induction on the length of the string $\boldsymbol{x} \in L$:
  - Assume $|\boldsymbol{x}| = 1$. Then, the only possible form for $\boldsymbol{x}$ is $\boldsymbol{x} = a$, where $a \in \Sigma$. Thus, $L = \{a\}$ which is the path notation of an ordered labeled tree.
  - Assume $|\boldsymbol{x}| \geq 3$ and that the claim holds for strings with lengths $1, 3, 5, \dots, |\boldsymbol{x}| - 2$. String $\boldsymbol{x}$ is of the form $\boldsymbol{x} = \boldsymbol{x}_1 1 \boldsymbol{w}$, where $\boldsymbol{x}_1 \in \Sigma$ and $\boldsymbol{w} \in \Sigma^+_{\leq k}$. Let $L_1 = \{\boldsymbol{w}\}$. Since all five conditions hold for L, it follows easily that they also hold for L_1. By induction hypothesis, we have that L_1 is the path notation of an ordered labeled tree. Thus,
$$L = \bigcup_{i=1}^{1} \{\boldsymbol{x}_1 i \boldsymbol{w} : \boldsymbol{w} \in L_1\},$$
which is the path notation of an ordered labeled tree. Hence, the claim holds for any L, where $|L| = 1$.

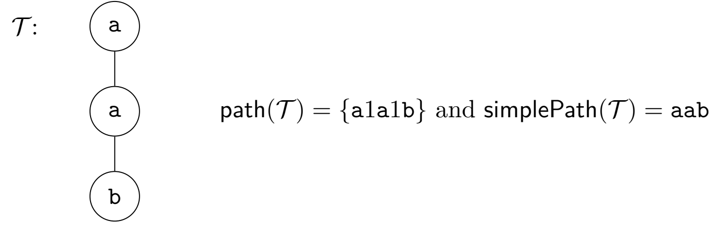


Figure 2.10 A linear ordered labeled tree $\mathcal{T}$ over $\{\mathtt{a}, \mathtt{b}\}$, its path notation, and its simple path notation.

- Assume $|L| \geq 2$ and that the claim holds for languages with lengths $1, 2, 3, \dots, |L| - 1$. Since the root condition and prefix condition hold for L, we get that $|\boldsymbol{x}| \neq 1$ for every $\boldsymbol{x} \in L$. Moreover, since the form condition holds for L, it follows that $|\boldsymbol{x}| \geq 3$ for every $\boldsymbol{x} \in L$. Our goal is to show that we can split L into $j \in \{1, \dots, k\}$ nonempty languages $L_1, \dots L_j$ where for each of them the five conditions hold. The splitting process is as follows: String $\boldsymbol{x} \in L$ goes to language L_i if $\boldsymbol{x}_2 = i$. Since the form condition and the sibling condition hold for L, we get that $\{\boldsymbol{x}_2 : \boldsymbol{x} \in L\} = \{1, 2, \dots, j\}$ for some $j \leq k$. Thus, there is always at least one string $\boldsymbol{x}$ such that $\boldsymbol{x}_2 = i$ for every $i \in \{1, \dots, j\}$. Thus, every L_i is nonempty. Now, for every L_i and every string $\boldsymbol{y} \in L_i$, we remove the first two symbols from $\boldsymbol{y}$. Since $|\boldsymbol{x}| \geq 3$ for every string $\boldsymbol{x} \in L$, it follows that no L_i contains the empty string. We can easily check that the five conditions hold for each L_i: The root condition holds for each L_i because L complies with the integrity condition. The remaining conditions hold for each L_i because they hold for L. By the induction hypothesis, we have that languages $L_1, \dots, L_j$ are path notations of ordered labeled trees $\mathcal{T}_1, \dots, \mathcal{T}_j$, respectively. Therefore,

$$L = \bigcup_{i=1}^{j} \{ai\boldsymbol{w} : \boldsymbol{w} \in \mathsf{path}(\mathcal{T}_i)\},$$

  where $a \in \Sigma$, which is the path notation of an ordered labeled tree. Hence, the claim holds. ◀

We use $L(\Sigma, k)$ to denote a string language over $\Sigma_{\leq k}$, which is the path notation of an ordered labeled tree over Σ.

The size of $L(\Sigma, k)$ corresponds to the number of leaves in the underlying tree. Thus, if the size of $L(\Sigma, k)$ is 1, then $L(\Sigma, k)$ represents a linear tree. We can simplify the path notation for a linear tree by omitting numbers. We call such a notation the *simple path notation* and denote it by $\mathsf{simplePath}(\mathcal{T})$, where $\mathcal{T}$ is a linear ordered labeled tree. See Figure 2.10.

The path notation can be used for both ranked and unranked trees. For an ordered ranked tree over Σ, where (Σ, rank) is a ranked alphabet, its path notation is a string language over $\Sigma_{\leq k}$, where k corresponds to the maximum rank of the symbols in Σ.

2.4.3 XML and XPath

In this section, we discuss the relationship between ordered labeled trees and XML documents. We also describe some constructs of an XML query language called XPath. There are two reasons why we include this section. First, ordered labeled trees can be taken as a formal model for data represented by XML documents. Thus, we can see XML as a linear tree notation. Second, querying an XML document can be viewed as a practical use case for the tree pattern matching problem, which we consider in Chapter 3, where we propose a classification for this problem. We note that this section does not aim to give a comprehensive description of XML and XPath but rather to describe the necessary features that we use later in the dissertation thesis.

XML is a markup language that defines a set of marks for encoding documents. The set of marks is not fixed and can be defined in various ways for each document. The key constructs of

```
<a>
  <b>
    <a></a>
    <b>
      <a></a>
    </b>
  </b>
  <a>
    <b></b>
  </a>
</a>
```

Figure 2.11 An XML document described as a sequence of tags.

an XML document are *tags*, *elements*, and *attributes*. A *tag* is a markup construct that begins with `<` (*start-tag*) or `</` (*end-tag*) and ends with `>`. An *element* is a logical document component that begins with a start tag and ends with a matching end-tag such as `<a>` and `</a>`. The symbols between the start-tag and end-tag are the element's content. An *attribute* is a markup construct consisting of a name-value pair separated by the equality sign within a start tag. In this dissertation thesis, we only consider the structure of XML documents and, therefore, ignore attributes and the content of elements. See an example of such an XML document in Figure 2.11.

The nested structure of XML documents gives a natural tree structure: each element represents a node in the tree; as the root, we consider the most outer element; and element-subelement relationships represent edges between the corresponding nodes. Although the order of the data stored in XML document might not be significant, its linear representation as a document naturally enforces order. Thus, we can view XML documents as ordered labeled trees. Note that an XML document described as a sequence of tags is a similar representation of an ordered labeled tree as the nested parenthesis notation. We can map the start-tag to a label with the consecutive open parenthesis and the end-tag to the corresponding closing parenthesis. The XML document described in Figure 2.11 can be seen as a representation of tree $\mathcal{T}_2$ illustrated in Figure 2.2.

Several languages, called XML schema languages, allow us to describe structure requirements on XML documents, such as DTD (Document Type Definition), which can be seen as an extended context-free grammar. Such a description, called a *schema*, can be taken as a description of a tree language that consists of trees representing all XML documents that satisfy a given schema.

Various query languages such as XPath have been designed to retrieve data from XML documents. In this dissertation thesis, we focus on three XPath constructs: node test for node labels, /-axis (child axis), and //-axis (descendant axis). Each query is a sequence of steps where each step consists of an axis and a node test. Specifically, given an alphabet Σ representing all possible element names in an XML document, we consider three types of queries:

- XPath(/) queries defined by the following regular grammar:

$$(\{S, A\}, \Sigma \cup \{/\}, \{S \to /A\} \cup \{A \to a \mid aS : a \in \Sigma\}, S)\,.$$

- XPath(//) queries defined by the following regular grammar:

$$(\{S, A\}, \Sigma \cup \{//\}, \{S \to //A\} \cup \{A \to a \mid aS : a \in \Sigma\}, S)\,.$$

- XPath(/, //) queries defined by the following regular grammar:

$$(\{S, A\}, \Sigma \cup \{/, //\}, \{S \to /A \mid //A\} \cup \{A \to a \mid aS : a \in \Sigma\}, S)\,.$$

Every string in a language generated by one of the three grammars represents a query that selects nodes from an XML document when evaluated. Evaluation occurs with respect to a

current node in the XML tree at which the processor is looking. An axis gives the direction to navigate from the current node and selects an initial node set. The node test is then used to filter the node set by specifying the desired element names. If a query starts with the /-axis, it selects the root (sets it as the current node); supposing that its label matches the label given by the consecutive node test in the query; otherwise, the empty set of nodes is returned as the answer. If a query starts with the //-axis, it selects all nodes with the label given by the consecutive node test, no matter where they are in the tree. For example, consider the XML document described in Figure 2.11 and its tree representation $\mathcal{T}_2$ illustrated in Figure 2.2. XPath query `//a/b//a` first looks for nodes labeled by `a` anywhere in the tree. Thus, selecting nodes $\mathtt{a}^1, \mathtt{a}^3, \mathtt{a}^5$, and $\mathtt{a}^6$. From there, it looks for children labeled by `b`. Thus, selecting nodes $\mathtt{b}^2$ and $\mathtt{b}^7$. Finally, it proceeds to select all descendants labeled by `a` and, hence, returning nodes $\mathtt{a}^3$ and $\mathtt{a}^5$ as the answer.

Chapter 3
Tree pattern matching

As stated earlier, the problem of tree pattern matching can be defined as the search for all occurrences of a pattern in an input tree. In the literature, tree pattern matching has several variants that differ in the type of tree considered, the definition of the pattern, and the specification when a pattern is considered to occur in an input tree. A tree can, for example, be free or rooted, ordered or unordered, and labeled or unlabeled. A pattern can be described by an ordinary tree, a finite set of trees, or a tree where nodes are labeled by special symbols such as wildcards and variables. It can also be described as a regular tree expression or a query using the syntax of query language, such as XPath. A pattern can be considered to occur in an input tree if, for example, a part of the input tree is the same as the pattern, or the matching process can be more tolerant and allow some errors.

In this dissertation thesis, we focus on the tree pattern matching problem for ordered labeled trees. In its most simplified form, the input of this problem is two ordered labeled trees, an input tree and a pattern tree, and the goal is to find all bottom-up subtrees in the input tree that are isomorphic to the pattern tree. In the literature, this problem is also known as the *bottom-up subtree isomorphism problem* [39, Section 4.2.3], [40] or *(bottom-up) subtree matching* [24], [30], [33], [38].

▶ **Problem 3.1** (The basic problem of tree pattern matching). Given ordered labeled trees $\mathcal{T}$ and $\mathcal{P}$, return the set of all nodes $v \in V(\mathcal{T})$ such that $\mathcal{P} \simeq \mathcal{T}/v$.

Another variant of the tree pattern matching problem that is frequently used, often called *tree template matching*, allows leaves of the pattern tree to be labeled by *subtree wildcards* [22, Section 3.1.2], [33, Section 2.2], [30], [36]. A node labeled by the subtree wildcard matches any bottom-up subtree of the input tree. We note that the subtree wildcard is also called *(sub)tree placeholder* [6] or *don't care symbol* [30], [36], [41] in the literature.

▶ **Problem 3.2** (Tree pattern matching problem with subtree wildcards). Let $\mathcal{T}$ and $\mathcal{P}$ be ordered labeled trees such that $\mathcal{P}$ contains k leaves labeled by subtree wildcard ν. Given $\mathcal{T}$ and $\mathcal{P}$, return the set of all nodes $v \in V(\mathcal{T})$ such that $\mathcal{P}[\mathcal{T}_1, \ldots, \mathcal{T}_k]_\nu \simeq \mathcal{T}/v$ for some ordered labeled trees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$.

There is also the notion of a *nonlinear tree pattern matching problem*, where nodes can be labeled by *subtree variables* [33], [68]. The difference between the wildcard and a variable is that the same tree must substitute occurrences of the same variable in the pattern. Apart from the basic variant of tree pattern matching and its variants with wildcards and variables, there are other variants of the tree pattern matching problem in the literature, such as *inexact* (or *approximate*) *tree pattern matching* [49], [50], [52], [69]. Moreover, some practical problems, such as querying XML documents, can be seen as variants of tree pattern matching problems [7], [8].

Although there exist various variants of the tree pattern matching problem, we have not found any unified naming standard for the problem. This can make the comparison of research results unnecessarily complex. In this chapter, we address this deficiency by proposing a classification of tree pattern matching problems.

Classification of pattern matching problems has already been used in the domain of strings. Melichar and Holub [70] classified the string pattern matching problem using six criteria: nature of the pattern, integrity of the pattern, number of patterns, way of matching, importance of symbols, and number of instances of the pattern. We can use these criteria to reference a particular variant of the string pattern matching problem using an abbreviation such as SFOECO problem, which represents the problem of exact string matching of one string: (S) the nature of the pattern is string, (F) we consider the full pattern in matching, (O) there is one pattern only, (E) we look for exact occurrences, (C) take care of all symbols (no don't care symbols are involved), and (O) there is one sequence of the pattern. In this chapter, we apply a similar approach to the problem of tree pattern matching.

This chapter is divided into four sections. First, we introduce our classification of tree pattern matching problems for ordered labeled trees. The following two sections are devoted to examples of using the classification as a unified naming standard for tree pattern matching problems and a discussion of some possible extensions of the classification. Finally, we conclude this chapter with a summary.

3.1 Classification of tree pattern matching problems

In this section, we present a classification of tree pattern matching problems for ordered labeled trees. We use the following criteria:

- structure of the pattern,
- nature of the pattern,
- integrity of the pattern,
- number of patterns,
- way of matching the pattern, and
- exactness of matching the pattern.

Each of the criteria is described in a separate section.

3.1.1 Structure of the pattern

As the first criterion, we consider the tree structure of the pattern. Specifically, we distinguish between two types of patterns:

(L) *linear patterns*, where patterns can be represented as linear trees, and

(N) *nonlinear patterns*, where representations of patterns are arbitrary trees (the degree of nodes can be greater than one).

We see the usefulness of creating a separate category for linear patterns because such problems tend to be easier to solve, and thus special methods for such patterns exist. A practical example of linear patterns are simple XPath queries such as /`a`/`b`/`c`. This query can be represented as a linear ordered labeled tree illustrated in Figure 3.1. We can also use a similar representation for XPath queries containing //-axis such as /`a`//`b`. However, these queries need special treatment because //-axis signifies that `a` is an ancestor of `b` (not necessarily its parent). This can be

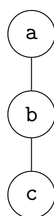


Figure 3.1 The linear ordered labeled tree that corresponds to XPath query `/a/b/c`.

handled, for example, by adding a special type of edges to a tree representation of the query or by adding a special wildcard label for nodes. We use the latter method and discuss it in the following section.

3.1.2 Nature of the pattern

As the second criterion, we consider the nature of the pattern. This criterion helps us define patterns that, when represented as trees, are allowed to contain special symbols as their node labels. In this category, we distinguish between the following variants of patterns:

(F) *fixed pattern*,

(LW) *pattern with label wildcards*,

(PW) *pattern with path wildcards*,

(SW) *pattern with subtree wildcards*,

(LV) *pattern with label variables*,

(PV) *pattern with path variables*, and

(SV) *pattern with subtree variables*.

A *fixed pattern* can be represented as an ordinary ordered labeled tree where each node has a specific, fixed label. We say that a fixed pattern *matches* a tree $\mathcal{T}$ if the tree representation of the pattern is isomorphic to $\mathcal{T}$. An example of such a pattern is the XPath query `/a/b/c` illustrated in Figure 3.1.

A *pattern with label wildcards* can have some of the nodes in its tree representation labeled by special symbol called *label wildcard*, the meaning of which is similar to the one used in string matching [71, Chapter 8]. The label wildcard matches all symbols of the alphabet representing possible node labels. An example of such a pattern is XPath query `/a/` $*$ `/b` that can be represented by a linear tree with three nodes: the root labeled by `a`, its (only) child labeled by the label wildcard $*$, and a leaf labeled by `b`. This pattern matches any linear tree with three nodes where the root is labeled by `a` and the leaf is labeled by `b`.

▶ **Definition 3.3** (Tree with label wildcards)**.** Let Σ be an alphabet and let $*$ be a special symbol, called *label wildcard*, such that $* \notin \Sigma$. The *tree with label wildcards* over $\Sigma \cup \{*\}$ is an ordered labeled tree over $\Sigma \cup \{*\}$.

▶ **Definition 3.4** (Function LWMatch)**.** Let Σ be an alphabet. Let $\mathcal{T}_1 = (V_1, E_1, r_1, \preceq_{S_1}, \mathsf{label}_1)$ be a tree with label wildcards over $\Sigma \cup \{*\}$. Let $\mathcal{T}_2 = (V_2, E_2, r_2, \preceq_{S_2}, \mathsf{label}_2)$ be an ordered labeled tree over Σ. A function LWMatch is defined for $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\mathsf{LWMatch}(\mathcal{T}_1, \mathcal{T}_2)$ is true if there exists a mapping $\varphi : V_1 \to V_2$ such that

- φ is a bijective mapping,

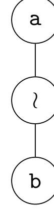


Figure 3.2 An example of a gapped tree over $\{\mathtt{a}, \mathtt{b}\} \cup \{\wr\}$.

- $\{u, v\} \in E_1$ if and only if $\{\varphi(u), \varphi(v)\} \in E_2$ for every pair of nodes $u, v \in V_1$,
- $\varphi(r_1) = r_2$,
- $u \preceq_{S_1} v$ if and only if $\varphi(u) \preceq_{S_2} \varphi(v)$ for every pair of nodes $u, v \in V_1$, and
- $\mathsf{label}_1(u) = \mathsf{label}_2(\varphi(u))$ for every node $u \in V_1$ where $\mathsf{label}_1(u) \neq *$.

Otherwise, $\mathsf{LWMatch}(\mathcal{T}_1, \mathcal{T}_2)$ is false.

A *pattern with path wildcards* have some of the internal nodes in its tree representation labeled by special symbol called *path wildcard*, the meaning of which is inspired by the XPath //-axis. We call an ordered labeled tree where internal nodes can be labeled by path wildcards a *tree with path wildcards* or a *gapped tree*.

▶ **Definition 3.5** (Tree with path wildcards, gapped tree)**.** Let Σ be an alphabet and let $\wr$ be a special symbol, called *path wildcard*, such that $\wr \notin \Sigma$. The *tree with path wildcards* (or *gapped tree*) over $\Sigma \cup \{\wr\}$ is an ordered labeled tree $\mathcal{T}$ over $\Sigma \cup \{\wr\}$ such that if $\mathsf{label}(v) = \wr$, where $v \in V(\mathcal{T})$, then v is an internal node and for each of its children $u \in \mathsf{children}(v)$ we have that $\mathsf{label}(u) \neq \wr$.

Note that it follows from the definition that there is at least one node in every gapped tree not labeled by the path wildcard. It also follows that the path wildcard can be used as the label of the root. In Figure 3.2, we illustrate an example of a gapped tree. This tree can be seen as a representation of XPath query /a//b.

We say that a gapped tree $\mathcal{T}$ *matches* another tree $\mathcal{T}'$ if $\mathcal{T}'$ is isomorphic to some extension of $\mathcal{T}$, which is obtained by substituting each occurrence of path wildcard for appropriate path of $\mathcal{T}'$ or the empty tree. We recall that the substitution of an internal node v for the empty tree corresponds to deleting v, which makes the children of v become children of $\mathsf{parent}(v)$; see Definition 2.28.

▶ **Definition 3.6** (Function PWMatch)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be a gapped tree over $\Sigma \cup \{\wr\}$ with $k \geq 0$ occurrences of path wildcard. Let $\mathcal{T}'$ be an ordered labeled tree over Σ. A function PWMatch is defined for $\mathcal{T}$ and $\mathcal{T}'$ such that $\mathsf{PWMatch}(\mathcal{T}, \mathcal{T}')$ is true if there exists a sequence $\mathcal{T}_1, \ldots, \mathcal{T}_k$, where $\mathcal{T}_i$ for each $i \in \{1, \ldots, k\}$ is either a linear ordered labeled tree over Σ or the empty tree such that $\mathcal{T}[\mathcal{T}_1, \ldots, \mathcal{T}_k]^{\wr} \simeq \mathcal{T}'$. Otherwise, $\mathsf{PWMatch}(\mathcal{T}, \mathcal{T}')$ is false.

For example, the gapped tree illustrated in Figure 3.2 matches every linear ordered labeled tree over $\{\mathtt{a}, \mathtt{b}\}$ where the root is labeled by a and the leaf is labeled by b. We illustrate several such trees in Figure 3.3.

Another possible nature of the pattern is the *pattern with subtree wildcards*. We can represent such a pattern by ordered labeled trees whose leaves are allowed to be labeled by *subtree wildcards*. In the literature, a notion *tree template* is sometimes used when at least one node is labeled by the subtree wildcard [23], [25], [33, Section 2.2], [30, Definition 64]. The subtree wildcard is usually represented by the symbol S or ν. In this dissertation thesis, we use the latter notation.

▶ **Definition 3.7** (Tree with subtree wildcards)**.** Let Σ be an alphabet and let ν be a special symbol, called *subtree wildcard*, such that $\nu \notin \Sigma$. The *tree with subtree wildcards* over $\Sigma \cup \{\nu\}$ is an ordered labeled tree $\mathcal{T}$ over $\Sigma \cup \{\nu\}$ such that if $\mathsf{label}(u) = \nu$, where $u \in V(\mathcal{T})$, then u is a leaf.

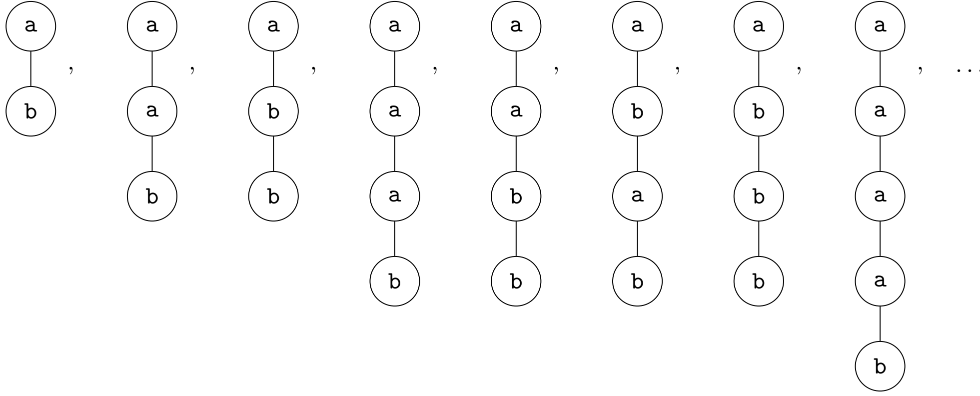


Figure 3.3 Ordered labeled trees over $\{\mathtt{a}, \mathtt{b}\}$ that match the gapped tree illustrated in Figure 3.2.

The subtree wildcard serves as a placeholder for an arbitrary subtree. That is, a tree with subtree wildcards $\mathcal{T}$ *matches* another tree $\mathcal{T}'$ if $\mathcal{T}'$ is isomorphic to some extension of $\mathcal{T}$, which is obtained by substituting each occurrence of subtree wildcard by an appropriate bottom-up subtree of $\mathcal{T}'$.

▶ **Definition 3.8** (Function SWMatch)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be a tree with subtree wildcards over $\Sigma \cup \{\nu\}$ with $k \geq 0$ occurrences of subtree wildcard. Let $\mathcal{T}'$ be an ordered labeled tree over Σ. A function SWMatch is defined for $\mathcal{T}$ and $\mathcal{T}'$ such that $\mathsf{SWMatch}(\mathcal{T}, \mathcal{T}')$ is true if there exists ordered labeled trees $\mathcal{T}_1, \ldots, \mathcal{T}_k$ over Σ such that $\mathcal{T}[\mathcal{T}_1, \ldots, \mathcal{T}_k]_\nu \simeq \mathcal{T}'$. Otherwise, $\mathsf{SWMatch}(\mathcal{T}, \mathcal{T}')$ is false.

The last three natures of patterns we consider in our classification are trees with variables. The difference between a wildcard and a variable is that the same label or tree must substitute for occurrences of the same variable in the pattern.

▶ **Definition 3.9** (Tree with label variables)**.** Let Σ_1 and Σ_2 be disjoint alphabets. Symbols from Σ_2 are called *label variables.* A *tree with label variables* over $\Sigma_1 \cup \Sigma_2$ is an ordered labeled tree over $\Sigma_1 \cup \Sigma_2$.

▶ **Definition 3.10** (Tree with path variables)**.** Let Σ_1 and Σ_2 be disjoint alphabets. Symbols from Σ_2 are called *path variables.* A *tree with path variables* over $\Sigma_1 \cup \Sigma_2$ is an ordered labeled tree $\mathcal{T}$ over $\Sigma_1 \cup \Sigma_2$ such that if $\mathsf{label}(v) \in \Sigma_2$, where $v \in V(\mathcal{T})$, then v is an internal node and for each of its children $u \in \mathsf{children}(v)$, we have that $\mathsf{label}(u) \neq \mathsf{label}(v)$.

Assume that there are no nodes $u, v \in \mathcal{T}$ labeled by the same symbol of Σ_2 such that $\mathsf{parent}(u) = v$.

▶ **Definition 3.11** (Tree with subtree variables)**.** Let Σ_1 and Σ_2 be disjoint alphabets. Symbols from Σ_2 are called *subtree variables.* A *tree with subtree variables* over $\Sigma_1 \cup \Sigma_2$ is an ordered labeled tree $\mathcal{T}$ over $\Sigma_1 \cup \Sigma_2$ such that if $\mathsf{label}(u) \in \Sigma_2$, where $u \in V(\mathcal{T})$, then u is a leaf.

A tree with subtree variables containing at least two nodes labeled by the same subtree variable is sometimes called a *nonlinear tree template* [33, Section 2.2].

Similarly, as for wildcards, we define what it means for a tree with variables to match another tree. For the tree with label variables, we adjust the conditions of tree isomorphism. In the case of the tree with path or subtree variables, we use the concurrent substitutions defined in Section 2.3.1.3.

▶ **Definition 3.12** (Function LVMatch)**.** Let Σ_1 and Σ_2 be disjoint alphabets. Let $\mathcal{T}_1 = (V_1, E_1, r_1, \preceq_{S_1}, \mathsf{label}_1)$ be a tree with label variables over $\Sigma_1 \cup \Sigma_2$. Let $\mathcal{T}_2 = (V_2, E_2, r_2, \preceq_{S_2}, \mathsf{label}_2)$ be an

ordered labeled tree over Σ_1. A function LVMatch is defined for $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\mathsf{LVMatch}(\mathcal{T}_1, \mathcal{T}_2)$ is true if there exists a mapping $\varphi : V_1 \to V_2$ such that

- φ is a bijective mapping,
- $\{u, v\} \in E_1$ if and only if $\{\varphi(u), \varphi(v)\} \in E_2$ for every pair of nodes $u, v \in V_1$,
- $\varphi(r_1) = r_2$,
- $u \preceq_{S_1} v$ if and only if $\varphi(u) \preceq_{S_2} \varphi(v)$ for every pair of nodes $u, v \in V_1$, and
- for every node $u \in V_1$, it holds that:
  - If $\mathsf{label}_1(u) \in \Sigma_1$, then $\mathsf{label}_1(u) = \mathsf{label}_2(\varphi(u))$.
  - If $\mathsf{label}_1(u) \in \Sigma_2$, then $\mathsf{label}_2(\varphi(u)) = \mathsf{label}_2(\varphi(v))$ for every node $v \in V_1$ where $\mathsf{label}_1(u) = \mathsf{label}_1(v)$.

Otherwise, $\mathsf{LVMatch}(\mathcal{T}_1, \mathcal{T}_2)$ is false.

▶ **Definition 3.13** (Function PVMatch)**.** Let Σ_1 and Σ_2 be disjoint alphabets. Let $\Sigma_2 = \{\wr_1, \dots, \wr_m\}$. Let $\mathcal{T}$ be a tree with path variables over $\Sigma_1 \cup \Sigma_2$. Let $\mathcal{T}'$ be an ordered labeled tree over Σ_1. A function PVMatch is defined for $\mathcal{T}$ and $\mathcal{T}'$ such that $\mathsf{PVMatch}(\mathcal{T}, \mathcal{T}')$ is true if there exists a sequence $\mathcal{T}_1, \dots, \mathcal{T}_m$, where $\mathcal{T}_i$ for each $i \in \{1, \dots, m\}$ is either a linear ordered labeled tree over Σ_1 or the empty tree such that $\mathcal{T}[\wr_1 \leftarrow \mathcal{T}_1, \dots, \wr_m \leftarrow \mathcal{T}_m] \simeq \mathcal{T}'$. Otherwise, $\mathsf{PVMatch}(\mathcal{T}, \mathcal{T}')$ is false.

▶ **Definition 3.14** (Function SVMatch)**.** Let Σ_1 and Σ_2 be disjoint alphabets. Let $\Sigma_2 = \{\nu_1, \dots, \nu_m\}$. Let $\mathcal{T}$ be a tree with subtree variables over $\Sigma_1 \cup \Sigma_2$. Let $\mathcal{T}'$ be an ordered labeled tree over Σ_1. A function SVMatch is defined for $\mathcal{T}$ and $\mathcal{T}'$ such that $\mathsf{SVMatch}(\mathcal{T}, \mathcal{T}')$ is true if there exists ordered labeled trees $\mathcal{T}_1, \dots, \mathcal{T}_m$ over Σ_1 such that $\mathcal{T}[\nu_1 \leftarrow \mathcal{T}_1, \dots, \nu_m \leftarrow \mathcal{T}_m] \simeq \mathcal{T}'$. Otherwise, $\mathsf{SVMatch}(\mathcal{T}, \mathcal{T}')$ is false.

3.1.3 Integrity of the pattern

Until now, we have considered matching the full pattern. Our interest was, for example, whether a pattern tree (as a whole) is isomorphic to another tree. However, we can sometimes be interested only in some parts of the pattern tree. For example, we can say that a pattern matches another tree if some subtree or rootpath of the pattern tree matches the tree. Whether we choose to match a full pattern or some of its parts can be used as another criterion for classifying tree pattern matching problems. We call this criterion the integrity of the pattern. This criterion includes the following variants:

(F) *full pattern*,

(SS) *subpattern: subtree*,

(SB) *subpattern: bottom-up subtree*,

(ST) *subpattern: top-down subtree*,

(SP) *subpattern: path*,

(SR) *subpattern: rootpath*, or

(SG) *subpattern: stringpath*.

3.1.4 Number of patterns

A pattern does not always have to be represented as a single tree. It can sometimes be described as a finite or infinite set of trees. If that is the case, then we say that a pattern *matches* tree $\mathcal{T}$ if at least one of the trees in the set representing the pattern matches $\mathcal{T}$. Using the number of trees representing a given pattern, we distinguish between three variants of tree pattern matching problems:

(O) *one*, in which the pattern is represented as a single tree,

(F) *a finite number greater than one*, in which a finite set of trees represents the pattern, and

(I) *an infinite number*, in which the pattern is represented by an infinite set of trees.

An infinite number of patterns can be given, for example, by a pattern in the form of a *regular tree expression*. Regular tree expressions can be seen as an extension of regular (string) expressions; for a more detailed exposition of the regular tree expressions, see Comen et al. [21, Section 2.2].

3.1.5 Way of matching the pattern

Until now, the criteria considered the description and interpretation of the pattern. For another criterion, we turn our attention to input trees. We distinguish between three variants of tree pattern matching problems according to which part of the input tree we match the pattern:

(S) *subtrees*,

(T) *top-down subtrees*, or

(B) *bottom-up subtrees*.

Subtrees, top-down subtrees, and bottom-up subtrees are the three main tree parts that we defined in Section 2.3.2. This criterion allows us to control to which of them in an input tree we match a given pattern. For example, consider a fixed, full pattern $\mathcal{P}$ represented as a single tree. If we choose subtrees as the way of matching, then there is an occurrence of $\mathcal{P}$ in a given input tree if $\mathcal{P}$ is isomorphic to some of its subtrees. Note that if the structure of $\mathcal{P}$ is linear, then matching it to subtrees corresponds to matching it to paths of the input tree. Similarly, matching patterns with the linear structure to top-down subtrees corresponds to matching it to rootpaths. For example, evaluation of XPath(/) queries corresponds to matching linear, fixed, full patterns represented as a single tree to top-down subtrees (rootpaths) of an input tree.

3.1.6 Exactness of matching the pattern

In some cases, a pattern rarely matches an input tree exactly. Pattern, input tree, or both can suffer from some undesirable corruption. Thus, we can classify tree pattern matching problems according to the exactness of matching into two variants:

(E) *exact matching*, in which a pattern must match parts of an input tree exactly, and

(I) *inexact matching*, in which we consider as occurrences all parts of an input tree where the pattern matches with up to a given maximum number of errors.

For inexact matching, we also need to specify which error model is used to define how different two trees are. For this purpose, a tree edit distance is often used. Apart from the 1-degree edit distance we defined in Section 2.3.1.2, there are other variants, and we discuss them briefly in Section 4.1.2.

Structure of pattern	Nature of pattern	Integrity of pattern	Number of patterns	Way of matching	Exactness of matching
L	F	F	O	S	E
N	(LW)	(SS)	F	T	I
	(PW)	(SB)	I	B	
	(SW)	(ST)			
	(LV)	(SP)			
	(PV)	(SR)			
	(SV)	(SG)			

Table 3.1 Overview of the SNINWE classification.

3.2 Describing tree pattern matching problems using the classification

We have defined six criteria to classify tree pattern matching problems. Table 3.1 gives an overview of the individual criteria and their possible values. Using the first letter of each criterion, we call this classification the *SNINWE classification* of tree pattern matching problems. In this section, we show how the classification can be used as a unified naming standard for tree pattern matching problems.

A particular tree pattern matching problem can be referenced using abbreviations of values in individual criteria. For example, Problem 3.1 corresponds to the NFFOBE tree pattern matching problem. As a second example, consider Problem 3.2. Using the SNINWE classification, we can refer to this problem as the N(SW)FOBE tree pattern matching problem. Moreover, we can give an alternative definition that uses the function SWMatch, see Problem 3.15.

▶ **Problem 3.15** (N(SW)FOBE tree pattern matching problem)**.** Given an ordered labeled tree $\mathcal{T}$ and a tree with subtree wildcards $\mathcal{P}$, the *N(SW)FOBE tree pattern matching problem* is to return the set of all nodes $v \in V(\mathcal{T})$ such that $\mathsf{SWMatch}(\mathcal{P}, \mathcal{T}/v)$ is true.

As the third example in this section, we define the NFFOBI tree pattern matching problem, a variant of inexact tree pattern matching problems. This problem is about finding all bottom-up subtrees in an input tree whose distance from the pattern tree is at most a given maximum number of errors. We revisit this problem in Chapter 5, where we present our automata-based solutions to this problem under the 1-degree edit distance.

▶ **Problem 3.16** (NFFOBI tree pattern matching problem)**.** Let Σ be an alphabet. Let $\mathcal{T}$ and $\mathcal{P}$ be ordered labeled trees over Σ. Let k be a non-negativity integer that represents the maximum number of errors allowed. Let f be a tree edit distance. Given $\mathcal{T}, \mathcal{P}, k$, and f, the *NFFOBI tree pattern matching problem* is to return the set of all nodes $v \in V(\mathcal{T})$ such that $f(\mathcal{P}, \mathcal{T}/v) \leq k$.

We give more examples of tree pattern matching problems described by the SNINWE classification when we discuss related work and introduce our results in tree indexing.

3.3 Possible extensions to the classification

The SNINWE classification covers many tree pattern matching problems. Using the values in individual criteria illustrated in Table 3.1, we can specify $2 \cdot 7 \cdot 7 \cdot 3 \cdot 3 \cdot 2 = 1764$ tree pattern matching problems. However, the SNINWE classification is not exhaustive. It can be extended by adding additional values to its criteria or by adding brand new criteria. To keep our classification clear, we chose to discuss some possible extensions in this section instead of trying to create

a universal complex classification. Possible extensions to existing criteria are, for example, as follows:

Structure of the pattern It could be helpful to consider structures other than linear and arbitrary trees. For example, we could know that patterns form trees in the shape of a star or that each node has a restricted number of children. This fact could be used when proposing solutions to the corresponding tree pattern matching problem; thus, it would be a reasonable extension of this criterion.

Nature of the pattern We can easily extend this criterion by allowing patterns to contain not only one type of special symbol but a combination, for example, a tree with path wildcards and subtree variables. We could also add parameters to path and subtree wildcards and variables that would set the minimum and the maximum number of nodes in a path or subtree that can be used as a substitution. For example, the path wildcard $\wr(1,3)$ would mean that we can only substitute it for a path containing one, two, or three nodes. Moreover, we could define special label wildcards or variables that match only a subset of an alphabet.

Integrity of matching Other tree parts, such as *leafpaths* (paths leading from a node in a tree to one of its leaves) or *ordered (top-down) subtrees* [39, Definition 4.5] could be used as other forms of subpatterns.

Way of matching We could match a given linear pattern to stringpaths of an input tree. Or we could use other tree parts that we have not defined in this dissertation thesis, such as the above-mentioned ordered (top-down) subtrees and leafpaths.

Exactness of matching Apart from exact and inexact matching, we could add *oracle matching* [28], [72] as another value. Oracle matching can report some false positives as matches.

We can also extend the classification by adding other criteria. One of the possible candidates is the *sequence of patterns* that Melichar and Holub [70] used in their classification of string pattern matching problems. In the context of strings, this criterion indicates that a pattern can be represented as a sequence of strings that must match consecutively. For example, a pattern $(\mathtt{abb}, \mathtt{ba})$ occurs in a given text if an occurrence of `abb` is followed by an occurrence of `ba` (with some possible gap in between). In the context of trees, we could allow a pattern to be represented as a sequence of trees. Such a pattern would match an input tree if, for example, the input tree contains the trees in the sequence consecutively as its bottom-up subtrees.

The so-called *online* and *offline* variants of pattern matching problems are often discussed in the literature. In the online variant, the input tree cannot be preprocessed, whereas in the offline version, it can, and we can thus build a data structure, known as *index*, on it. The index can then be used to speed up the searching phase. Whether we consider the online or offline variant can be added as another classification criterion.

We can also classify tree pattern matching problems according to their output specification. The output can take several forms similar to the output forms of string pattern matching problems. Crochemore et al. [71, Page 20] give an example of three output forms for string pattern matching: a boolean value encoding whether a pattern occurs or does not in a text, a binary string of the same length of the input text encoding right positions of occurrences, and a set of right or left positions of occurrences of a pattern in a text. Similarly, a tree pattern matching problem can also be formulated as a decision problem where the aim is only to test whether a pattern occurs in an input tree or not without specifying the positions of the possible occurrences. The advanced version of the output we have used so far is to formulate tree pattern matching problems as search problems, which entails searching for all pattern occurrences in the input tree. The output can be a set of nodes representing the positions of occurrences or even take the form of the binary string encoding the occurrences if an input tree is represented as a string using a linear tree notation. Moreover, the output of the search problem can also only be the number of occurrences and not the occurrences themselves, or the output can be only the first or last occurrence.

Even more variants for the output are possible in the case of inexact tree pattern matching. We can formulate such a problem as an optimization problem where the aim is to report in some sense an optimal occurrence. Concerning the desired output, we can distinguish between the following variants of inexact tree pattern matching problems:

Bounded matches Given a desired maximum distance, all matches with a distance less than or equal to this distance are considered.

Best match Only the best match with the least distance is considered.

Top k matches Only the best k matches for a given tree edit distance are considered.

3.4 Summary

In this chapter, we addressed the absence of a unified naming standard for tree pattern matching problems by proposing a classification called the SNINWE classification It categorizes tree pattern matching problems according to six criteria: the structure of the pattern, the nature of the pattern, the integrity of the pattern, the number of patterns, the way of matching the pattern, and the exactness of matching the pattern. For designing the classification, we applied a similar approach that Melichar and Holub [70] used to classify string pattern matching problems.

The SNINWE classification can be used to reference various tree pattern matching problems using abbreviations such as LFFOSE problem, which corresponds to a tree pattern matching problem where

L the representation of a pattern is a Linear tree,

F all nodes in the pattern tree are labeled by specific, Fixed labels (no nodes are labeled by special symbols such as wildcards or variables),

F the Full pattern is considered during the matching,

O the pattern can be represented as One tree,

S we search for occurrences in the Subtrees of a given input tree, and

E we search for Exact occurrences.

Although the SNINWE classification covers many tree pattern matching problems, it is not exhaustive. In this chapter, we also discussed its possible extensions, which can be done by adding values to existing criteria or introducing new criteria. We refer to the SNINWE classification throughout this dissertation thesis when we discuss related work and introduce our results in inexact tree pattern matching and tree indexing.

Chapter 4

Previous results and related work

The main scope of this dissertation thesis is pattern matching in ordered labeled trees. Since there is a rather large body of work devoted to this problem, we do not attempt to list all existing results here. Instead, we focus on the most relevant work in the domain of inexact tree pattern matching and tree indexing, the two problems for which we propose automata-based solutions in the following two chapters. Moreover, since this dissertation thesis focuses on the string automata approach, we pay more attention to existing methods that are based on string automata.

This chapter consists of two sections. First, we focus on the problem of inexact pattern matching. Specifically, we discuss existing matching algorithms that preprocess a given pattern and not the input data. Second, we pay attention to the problem of indexing, where algorithms preprocess the input data and build a data structure known as an index. In both of these sections, we discuss relevant work in the domain of trees and strings. We mention the string results because many stringology algorithms and principles can be, with some care, adapted to handle trees represented as strings. For strings, we describe mainly established work that is the basis of our automata-based methods proposed in the following two chapters.

4.1 Inexact pattern matching

Pattern matching is the problem of finding all occurrences of a pattern in the input data. However, sometimes the pattern, the input data, or both can be subject to deformation or corruption, which means that matching algorithms need to be designed to be tolerant and allow the presence of some errors. In other words, the goal is to locate all occurrences within the input data that are similar to the pattern. This problem is called inexact (or approximate) pattern matching.

This section gives an overview of previous results for inexact pattern matching. Specifically, we focus on online searching when only a pattern can be preprocessed and not the input data to build an index. First, we pay attention to the inexact string pattern matching problem. Then, we describe existing approaches to inexact pattern matching in trees.

4.1.1 Strings

The problem of inexact string pattern matching is about searching for occurrences of a given pattern in other strings (or *texts*, to be less formal) that are similar to a given pattern under some similarity measure. Various variants of the inexact string pattern matching problem are discussed in the literature, and they differ mainly by the error model used [73].

An error model defines how different two strings are, and a well-known one is the so-called *edit distance*. The notion of distance between two strings was defined by Wagner and Fisher in 1974 [43]. They considered three elementary edit operations:

- *renaming* (also called *substitution* or *replacement*) of a symbol by a different symbol,
- *deletion* of a symbol, and
- *insertion* of a symbol.

This most prominent distance for strings is also known as the *Levenshtein distance* [74].

Similarly, as in the case of 1-degree edit operations discussed in Section 2.3.1.2, a *cost* can be assigned to the elementary edit operations mentioned above using an alphabet with the blank symbol equipped with a cost function. If all operations cost one, we speak of the *simple* Levenshtein distance. The problem of computing the simple Levenshtein distance between two strings is thus to find the minimum number of operations to transform one string into the other. The notions of an *edit script*, the *cost of an edit script*, the *length of an edit script*, an *optimal edit script*, and a *shortest edit script* that we established in Section 2.3.1.2 can be used analogously in the context of strings.

▶ **Definition 4.1** (simple Levenshtein distance)**.** Let Σ be an alphabet. Given two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ over Σ, the *simple Levenshtein distance* is a function $\mathsf{sLev} : \Sigma^* \times \Sigma^* \to \mathbb{N}_0$ such that $\mathsf{sLev}(\boldsymbol{x}, \boldsymbol{y})$ is the length of a shortest edit script between $\boldsymbol{x}$ and $\boldsymbol{y}$.

▶ **Definition 4.2** (Levenshtein distance)**.** Let Σ_λ be an alphabet with the blank symbol λ equipped with a cost function. Given two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ over Σ, the *Levenshtein distance* is a function $\mathsf{Lev} : \Sigma^* \times \Sigma^* \to \mathbb{R}$ such that $\mathsf{Lev}(\boldsymbol{x}, \boldsymbol{y})$ is the cost of an optimal edit script between $\boldsymbol{x}$ and $\boldsymbol{y}$.

The Levenshtein distance is not the only distance between strings that has been used [44, Chapter 11]. Other well-known string edit distances are, for example, the *Hamming distance* [75], the *Damerau distance* [76], the *longest common subsequence distance* [77], [78], or the *episode distance* [79].

In this section, we narrow our focus to the previous work on the inexact string pattern matching under the simple Levenshtein distance, also known as the problem of *string matching with k differences* [16], [73], [80], [81]. Particularly, we pay attention to the results on which we build in Chapter 5, where we introduce our automata-based approach to the inexact pattern matching in trees.

▶ **Problem 4.3** (Inexact string pattern matching under the simple Levenshtein distance)**.** Let Σ be an alphabet. Let $\boldsymbol{p}$ and $\boldsymbol{t}$ be strings over Σ. Let $k \geq 0$ represent the maximum number of errors allowed. Given $\boldsymbol{p}, \boldsymbol{t}$, and k, the *inexact string pattern matching problem under the simple Levenshtein distance* is to return the set of all positions j in $\boldsymbol{t}$ for which there exists j' such that $\mathsf{sLev}(\boldsymbol{p}, \boldsymbol{t}_{j' \dots j}) \leq k$.

There are currently four approaches to the problem of inexact string pattern matching under the (simple) Levenshtein distance: dynamic programming [43], [82] [12, Section 6.2.2.3], automata [16], [46] [12, Section 2.2.3.2], bit-parallelism [47], [48], [83] [12, Section 6.3.2.3], and filtering [48]. In the rest of this section, we pay attention to the results of the first two approaches. For an overview of all approaches, we refer the reader to the survey by Navarro [73].

4.1.1.1 Dynamic programming approach

The dynamic programming approach (see, for example, Cormen et al. [62, Chapter 15]) is the oldest approach to the problem of inexact string pattern matching under the (simple) Levenshtein distance. It builds on the basic algorithm that computes the simple Levenshtein distance; see Algorithm 4.4 and Figure 4.1. The computation of the (simple) Levenshtein distance between two strings can be attributed to Wagner and Fisher [43] or Needleman and Wunsch [77].

D	j	0	1	2	3	4	5	6	7
i	-	-	a	d	c	a	b	c	a
0	-	0	1	2	3	4	5	6	7
1	a	1	0	1	2	3	4	5	6
2	d	2	1	0	1	2	3	4	5
3	b	3	2	1	1	2	2	3	4
4	b	4	3	2	2	2	2	3	4
5	c	5	4	3	2	3	3	2	3
6	a	6	5	4	3	2	3	3	2

Figure 4.1 Computing the simple Levenshtein distance between strings `adbbca` and `adcabca` using Algorithm 4.4. The blue entries show the paths to the final result.

▶ **Algorithm 4.4** (The dynamic programming approach to computing the simple Levenshtein distance [43]). Let Σ be an alphabet. Given strings $\boldsymbol{x}$ and $\boldsymbol{y}$ over Σ, the algorithm computes $\mathsf{sLev}(\boldsymbol{x}, \boldsymbol{y})$. The principal internal data structure is a two-dimensional array $\boldsymbol{D}$ in which the dimensions have ranges 0 to $|\boldsymbol{x}|$ and 0 to $|\boldsymbol{y}|$, respectively. By $\boldsymbol{D}_{i,j}$, where $i \in \{0, \dots, |\boldsymbol{x}|\}$ and $j \in \{0, \dots |\boldsymbol{y}|\}$, we denote the value at row i and column j. When the array is filled, $\boldsymbol{D}_{i,j}$ contains $\mathsf{sLev}(\boldsymbol{x}_{1\dots i}, \boldsymbol{y}_{1\dots j})$. Thus, $\boldsymbol{D}_{|\boldsymbol{x}|,|\boldsymbol{y}|}$ contains $\mathsf{sLev}(\boldsymbol{x}, \boldsymbol{y})$.

1. (Initialize array.)
   a. Set $\boldsymbol{D}_{0,0} \leftarrow 0$.
   b. For each position i in $\boldsymbol{x}$, set $\boldsymbol{D}_{i,0} \leftarrow i$.
   c. For each position j in $\boldsymbol{y}$, set $\boldsymbol{D}_{0,j} \leftarrow j$.
2. (Fill array.) For each position j in $\boldsymbol{y}$ and i in $\boldsymbol{x}$:
   a. The cost of renaming $\boldsymbol{x}_i$ to $\boldsymbol{y}_j$ (or match) is $c_1 \leftarrow \begin{cases} \boldsymbol{D}_{i-1,j-1} & \text{if } \boldsymbol{x}_i = \boldsymbol{y}_j, \\ \boldsymbol{D}_{i-1,j-1} + 1 & \text{otherwise.} \end{cases}$
   b. The cost of deleting $\boldsymbol{x}_i$ from $\boldsymbol{x}$ is $c_2 \leftarrow \boldsymbol{D}_{i-1,j} + 1$.
   c. The cost of inserting symbol $\boldsymbol{y}_j$ into $\boldsymbol{x}$ is $c_3 \leftarrow \boldsymbol{D}_{i,j-1} + 1$.
   d. Set $\boldsymbol{D}_{i,j} \leftarrow \mathsf{min}(c_1, c_2, c_3)$.
3. (Return.) Return $\boldsymbol{D}_{|\boldsymbol{x}|,|\boldsymbol{y}|}$.

A slightly different variant of Algorithm 4.4 also appears in the literature [73, Section 5.1.1], [16]. In this variant, the observation that $\boldsymbol{D}_{i-1,j} + 1$ or $\boldsymbol{D}_{i,j-1} + 1$ cannot be smaller than $\boldsymbol{D}_{i-1,j-1}$ [84], [71, Section 8.2] is used and Step 2 is changed as follows:

2. (Fill array.) For each position j in $\boldsymbol{y}$ and i in $\boldsymbol{x}$: If $\boldsymbol{x}_i = \boldsymbol{y}_j$, then $\boldsymbol{D}_{i,j} \leftarrow \boldsymbol{D}_{i-1,j-1}$. Otherwise, set $\boldsymbol{D}_{i,j} \leftarrow 1 + \mathsf{min}(c_1, c_2, c_3)$, where
   - $c_1 \leftarrow \boldsymbol{D}_{i-1,j-1}$ is the cost of renaming $\boldsymbol{x}_i$ to $\boldsymbol{y}_j$,
   - $c_2 \leftarrow \boldsymbol{D}_{i-1,j}$ is the cost of deleting $\boldsymbol{x}_i$ from $\boldsymbol{x}$, and
   - $c_3 \leftarrow \boldsymbol{D}_{i,j-1}$ is the cost of inserting symbol $\boldsymbol{y}_j$ into $\boldsymbol{x}$.

$\boldsymbol{D}$	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	-	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
2	d	2	1	0	1	1	1	2	1	1	1	1	0	1	2	2	1
3	b	3	2	1	1	2	1	2	2	2	1	2	1	0	1	2	2
4	b	4	3	2	2	2	2	2	3	3	2	2	2	1	0	1	2
5	c	5	4	3	2	3	3	2	3	4	3	3	3	2	1	0	1
6	a	6	5	4	3	2	3	3	2	3	4	3	4	3	2	1	0

Figure 4.2 Search for $\boldsymbol{p} = \texttt{adbbca}$ in $\boldsymbol{t} = \texttt{adcabcaabadbbca}$ with $k = 2$ under the simple Levenshtein distance. The occurrences are found in $\boldsymbol{t}$ at positions 4, 7, 13, 14, and 15 which correspond to end positions of occurrences. The blue entries show the paths to the final result.

Since the size of the array is $(|\boldsymbol{x}|+1)\cdot(|\boldsymbol{y}|+1)$ and each value can be computed in constant time, Algorithm 4.4 works in time $\mathcal{O}(|\boldsymbol{x}| \cdot |\boldsymbol{y}|)$. Assuming $|\boldsymbol{x}| < |\boldsymbol{y}|$, the algorithm can be implement to use $\mathcal{O}(|\boldsymbol{x}|)$ space because for each pair $i, j \geq 1$, the value $\boldsymbol{D}_{i,j}$ depends only on the values at the three neighbor positions: $\boldsymbol{D}_{i-1,j-1}, \boldsymbol{D}_{i,j-1}$, and $\boldsymbol{D}_{i-1,j}$. Thus, two columns are sufficient for computation.

Several researchers aimed to improve Algorithm 4.4 since 1970s. For example, Masek and Paterson [85] presented an algorithm that works in $\mathcal{O}\big(n^2/\log n\big)$ time for strings $\boldsymbol{x}$ and $\boldsymbol{y}$ of the same length equal to n. In 2015, Backurs and Indyk [86] provided evidence that the near-quadratic searching time for the problem of computing the simple Levenshtein distance might be tight.

Given strings $\boldsymbol{x}$ and $\boldsymbol{y}$, Algorithm 4.4 computes $\mathsf{sLev}(\boldsymbol{x}, \boldsymbol{y})$. To obtain the corresponding edit script between $\boldsymbol{x}$ and $\boldsymbol{y}$, we can perform the computation by tracking back the array from $\boldsymbol{D}_{|\boldsymbol{x}|,|\boldsymbol{y}|}$ to $\boldsymbol{D}_{0,0}$ as illustrated in Figure 4.1. Note that multiple paths may exist. This computation takes $\mathcal{O}(|\boldsymbol{x}| + |\boldsymbol{y}|)$ additional time and space. For details, see Algorithm Y by Wagner and Fischer [43] or Section 11.3.3 by Gusfield [87].

The algorithm for computing the (simple) Levenshtein distance was converted into a searching algorithm by Sellers [82]. Assuming that $\boldsymbol{x}$ is the pattern and $\boldsymbol{y}$ is the input text, the algorithm can locate all substrings $\boldsymbol{y}'$ of $\boldsymbol{y}$ such that $\mathsf{sLev}(\boldsymbol{x}, \boldsymbol{y}') \leq k$, where k is a given maximum number of errors. The algorithm is basically the same as Algorithm 4.4. The only difference is that any position in the input text is now a potential start, which corresponds to initializing the values of the first row of the array to zero. The positions of the occurrences can then be found in the last row where the values are not greater than k. Specifically, given strings $\boldsymbol{x}$ (the pattern) and $\boldsymbol{y}$ (the text) and $k \geq 0$, Algorithm 4.4 can be converted into a searching algorithm as follows:

- Change Step 1c as follows: For each position j in $\boldsymbol{y}$, set $\boldsymbol{D}_{0,j} \leftarrow 0$.
- Change Step 3 (Report occurrences.) as follows: For each position j in $\boldsymbol{y}$: If $\boldsymbol{D}_{|\boldsymbol{x}|,j} \leq k$, output j.

From now on, we use $\boldsymbol{p}$ and $\boldsymbol{t}$ instead of $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively, to denote the pattern and the input text. We illustrate the dynamic programming approach to the problem of inexact string pattern matching under the simple Levenshtein distance in Figure 4.2.

Given $\boldsymbol{p}, \boldsymbol{t}$, and k, the searching algorithm mentioned above works in time $\mathcal{O}(|\boldsymbol{p}| \cdot |\boldsymbol{t}|)$ since the array of size $(|\boldsymbol{p}| + 1) \cdot (|\boldsymbol{t}| + 1)$ is filled and each value can be computed in constant time. Assuming that $|\boldsymbol{p}| < |\boldsymbol{t}|$, the computation can be performed in $\mathcal{O}(|\boldsymbol{p}|)$ space for similar reasons as in the case of Algorithm 4.4.

The dynamic programming approach turned out to be very flexible. For example, it can be adapted to the Levenshtein distance (when operations do not have the unit cost). The main idea behind this adaptation lies in changing the value added when an edit operation is applied to the current position. When matching under the simple Levenshtein distance, we always add one. In the case of non-unit cost operations, the operation cost is added instead. See, for example, Wagner and Fisher [43] or Section 11.5 by Gusfield [87] for details. The algorithm can also be modified for another set of edit operations [84] such as to allow transpositions (two consecutive symbols can be swapped; each symbol can be involved only in one transpose) [12, Section 6.2.2.4]. By considering only substitutions, the algorithm computes the Hamming distance [88].

The two-dimensional array that is filled during the matching under the simple Levenshtein distance has some properties such as that two adjacent values on a column, row, and diagonal differ by at most one [84], [71, Section 8.2]. Moreover, Ukkonen [46] observed that all the values in the array that are larger than $k+1$ can be replaced by $k+1$ without affecting the output of the search. These properties have been used to design more efficient algorithms that process the text in time $\mathcal{O}(k \cdot |\boldsymbol{t}|)$ [89], [90].

4.1.1.2 Automata approach

An alternative approach to the problem of inexact string pattern matching under the simple Levenshtein distance is to use a finite automaton [9], [16], [46]–[48], [12, Section 2.2.3.2]. Using a deterministic finite automaton, all inexact occurrences of pattern $\boldsymbol{p}$ in text $\boldsymbol{t}$ can be found in time $\mathcal{O}(|\boldsymbol{t}|)$. The disadvantage of this approach, however, is its space complexity.

The finite automaton for the problem of inexact string pattern matching under the simple Levenshtein distance can be constructed using the so-called *Levenshtein automaton* that accepts the set of all strings which are at most at distance k to $\boldsymbol{p}$, where $\boldsymbol{p}$ is the pattern and k represent the maximum number of errors. The nondeterministic Levenshtein automaton has a regular structure; see Algorithm 4.5 and Figure 4.3. The automaton has $(k+1) \cdot (|\boldsymbol{p}|+1)$ states and can be built in time $\mathcal{O}(k \cdot |\boldsymbol{p}|)$.

▶ **Algorithm 4.5** (Construction of the Levenshtein automaton)**.** Let Σ be an alphabet. Let $\boldsymbol{p}$ be a string over Σ. Let $k \geq 0$. Given $\boldsymbol{p}$ and k, the algorithm constructs an ε-NFA $\mathcal{M}$ such that $L(\mathcal{M}) = \{\boldsymbol{p}' : \mathsf{sLev}(\boldsymbol{p}, \boldsymbol{p}') \leq k\}$.

1. (Define the set of states.) Each state is labeled by i^l, where $i \in \{0, \ldots, |\boldsymbol{p}|\}$ and $l \in \{0, \ldots, k\}$. Specifically, states are labeled so they can be arranged into rows and columns as illustrated in Figure 4.3. We say that a state labeled by i^l it is at *depth* i and on *level* l.

$$\text{Set } Q \leftarrow \left\{i^l : 0 \leq i \leq |\boldsymbol{p}| \wedge 0 \leq l \leq k\right\}.$$

2. (Define the start state.) Set $q_0 \leftarrow 0^0$.

3. (Define the set of final states.) Every state at depth $|\boldsymbol{p}|$ is final.

$$\text{Set } F \leftarrow \left\{|\boldsymbol{p}|^l : 0 \leq l \leq k\right\}.$$

4. (Add transitions indicating match.) For each position $i \in \{1, \ldots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \ldots, k\}$, set $\delta\left((i-1)^l, \boldsymbol{p}_i\right) \leftarrow \left\{i^l\right\}$.

5. (Add transitions for renaming operations.) For each position $i \in \{1, \ldots, |\boldsymbol{p}|\}$, number of errors $l \in \{0, \ldots, k-1\}$, and symbol $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$, set $\delta\left((i-1)^l, a\right) \leftarrow \left\{i^{l+1}\right\}$.

6. (Add transitions for insertion operations.) For each $i \in \{0, \ldots, |\boldsymbol{p}|\}$, number of errors $l \in \{0, \ldots, k-1\}$, and symbol $a \in \Sigma$, set $\delta\left(i^l, a\right) \leftarrow \delta\left(i^l, a\right) \cup \left\{i^{l+1}\right\}$.

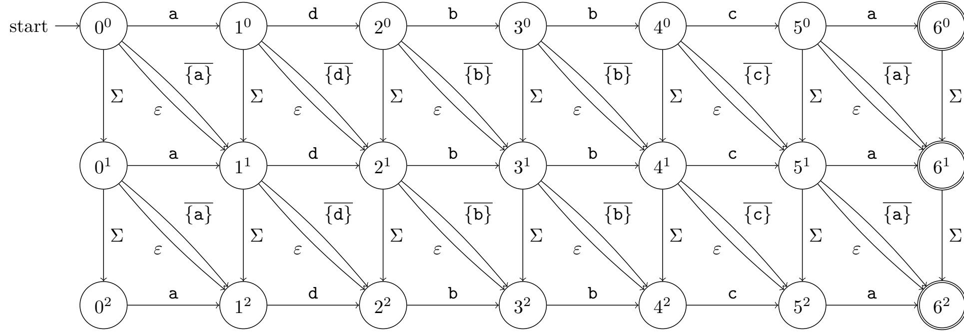


Figure 4.3 Levenshtein automaton for $\boldsymbol{p} = \texttt{adbbca}$ and $k = 2$. Horizontal transitions represent matches. Each of the remaining transitions represents using an edit operation. Vertical transitions correspond to insertion operation, diagonal ε-transitions represent deletion operation, and the remaining diagonal transitions represent renaming operation.

7. (Add transitions for deletion operations.) For each position $i \in \{1, \ldots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \ldots, k-1\}$, set $\delta\left((i-1)^l, \varepsilon\right) \leftarrow \left\{i^{l+1}\right\}$.
8. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

Given a string $\boldsymbol{p}$ over alphabet Σ and $k \geq 0$, the Levenshtein automaton for $\boldsymbol{p}$ and k can be turned into a matching automaton by simply adding a loop transition to its start state for each symbol of Σ. To locate all occurrences of $\boldsymbol{p}$ in given text $\boldsymbol{t}$ with up to k errors, it is sufficient to run the automaton on $\boldsymbol{t}$ and report an occurrence every time a final state is active. The position in the input text corresponds to the end position of the occurrence. Note that more than one final states can be active at the same time; the distance between $\boldsymbol{p}$ and its occurrence in $\boldsymbol{t}$ corresponds to the level of the topmost active final state.

The automata-based approach is similarly flexible as the dynamic programming approach. The ε-NFA can be adapted to different costs of operations by changing the level of the target state of transitions that correspond to non-unit cost operations. For example, if deletion costs 2 instead of 1, we make the ε-transitions move from a state at level l to a state at level $l+2$ instead of $l+1$. The automaton can also be adapted for other edit operations such as the Hamming distance [9], [88] or to allow transpositions by introducing auxiliary states [12, Section 2.2.3.3]. The ε-NFA can also be modified to the problem of inexact string pattern matching under distances based on an *ordered alphabet* in which the order of symbols matters [12, Section 2.2.4].

There is a connection between the ε-NFA and the dynamic programming approach to the inexact string pattern matching problem under the simple Levenshtein distance. As observed, for example, by Melichar [16], [12, Theorem 6.11], the two-dimensional array computed in the dynamic programming algorithm simulates the ε-NFA in the following way:

- Value $\boldsymbol{D}_{i,j} \leq k$ corresponds to the level of the topmost active state in depth i of the ε-NFA and step j of the run of the ε-NFA.
- Value $\boldsymbol{D}_{i,j} > k$ signals that there is no active state in depth i of the ε-NFA and step j of the run of the ε-NFA.

The ε-NFA can be made deterministic (by eliminating ε-transitions and using subset construction) to obtain linear search time in the length of a given input text. However, the practicality of this approach is reduced due to the space requirements for the DFA.

Melichar [12, Section 2.4] showed that the number of states of the DFA that is obtained from the ε-NFA for inexact matching under the simple Levenshtein distance is $\mathcal{O}\left(|\Sigma|^k \cdot |\boldsymbol{p}|^{k+1}\right)$, where $\boldsymbol{p}$ is the pattern over alphabet Σ and k is the maximum number of errors allowed. To prove this state complexity, Melichar used some existing results from dictionary matching.

Given a finite set D of nonempty strings over alphabet Σ called *dictionary* and a string $\boldsymbol{t} \in \Sigma^*$, the problem of *dictionary matching* is to locate all occurrences of strings of D in $\boldsymbol{t}$ [71, Chapter 2]. The *dictionary automaton* is then a finite automaton accepting language $\Sigma^* D$.

Given a dictionary D, let $\mathsf{pref}(D)$ be the set of all strings such that each string is a prefix of some string in D. Clearly, $|\mathsf{pref}(D)|$ is at most $1 + \sum_{\boldsymbol{x} \in D} |\boldsymbol{x}|$. The deterministic dictionary automaton for D can be constructed according to Proposition 5.1 given by Crochemore et al. [10, Section 5.2]. The set of states of this automaton corresponds $\mathsf{pref}(D)$. Later, Melichar [12, Theorem 2.34] proved the equivalence of this automaton to the DFA that is created using three steps:

1. Create a deterministic finite automaton with a tree-like structure that accepts all strings in D. The set of states of this automaton corresponds to $\mathsf{pref}(D)$.
2. Add a loop transition to the start state of the automaton created in Step 1 for all alphabet symbols.
3. Use subset construction to convert the NFA from Step 2 to an equivalent DFA.

Moreover, Melichar discussed that the consequence of the proof is that during the transformation of the NFA to DFA the number of states does not increase. Therefore, the DFA has $1 + \sum_{\boldsymbol{x} \in D} |\boldsymbol{x}|$ states in the worst case.

Clearly, a deterministic dictionary automaton can be built from any finite automaton accepting a finite language D by adding a loop transition to its start state for all symbols of the alphabet and using subset construction. Melichar [12, Theorem 2.35] proved that the number of states of such a DFA is also $1 + \sum_{\boldsymbol{x} \in D} |\boldsymbol{x}|$ states in the worst case.

▶ **Theorem 4.6** (Upper bound for the number of states of the deterministic dictionary automaton [12])**.** *Let D be a dictionary. Let $\mathcal{M}$ be an NFA accepting D. Let $\mathcal{M}'$ be the NFA obtained from $\mathcal{M}$ by adding a loop transition to its start state for all symbols of the alphabet. Then, the number of states of DFA obtained from $\mathcal{M}$ by the subset construction is*

$$\mathcal{O}\Big(\sum_{\boldsymbol{x} \in D} |\boldsymbol{x}|\Big).$$

Using these results from dictionary matching, Melichar [12, Theorem 2.42] proved that the DFA obtained from the ε-NFA for the problem of inexact string pattern matching under the simple Levenshtein distance has $\mathcal{O}\big(|\Sigma|^k \cdot |\boldsymbol{p}|^{k+1}\big)$ states, where $\boldsymbol{p}$ is the pattern over alphabet Σ and k is the maximum number of errors allowed.

In addition to obtaining the DFA from the ε-NFA, other researchers have studied the direct construction of the DFA. For example, Ukkonen [46] proposed the idea of a construction of the DFA directly from the dynamic programming array. Each state of the DFA corresponds to a possible column which can occur in the array. Ukkonen proved that the space complexity of this DFA is $\mathcal{O}\big(|\Sigma| \cdot \min\big(3^{|\boldsymbol{p}|}, 2^k \cdot |\Sigma|^k \cdot |\boldsymbol{p}|^{k+1}\big)\big)$, where $\boldsymbol{p}$ is the pattern over alphabet Σ and k is the maximum number of errors allowed. Later, Melichar [16] refined the bound to $\mathcal{O}\big((k+2)^{|\boldsymbol{p}|-k} \cdot (k+1)! \cdot \min(|\Sigma|, |\boldsymbol{p}|+1)\big)$.

Because of the space requirement of the DFA, Navarro [91] studied the computation of the automaton in lazy form; that is, only the states actually reached when the text is read are generated. An alternative approach is to use bit-parallel simulation of the (ε-)NFA instead of turning it into the equivalent DFA [47], [48], [83].

4.1.2 Trees

The problem of inexact tree pattern matching is a direct extension of inexact string pattern matching. Similarly, as in the case of strings, this problem is connected with the well-known tree edit distance problem, which is about measuring the similarity between two trees.

There are various types of tree edit distance based on the set of edit operations allowed and the type of trees considered [44, Chapter 15.4], [45]. We recall that in Section 2.3.1.2, we described the 1-degree edit distance introduced in 1977 by Selkow [42], where the set of elementary edit operations consists of node relabeling, leaf insertion, and leaf deletion. A natural extension of this distance is called *degree-2 distance* [92], where one can delete either a leaf or a node with one child; the insertion operation is then the complement to the deletion operation. One of the most studied edit distances between ordered labeled trees was introduced two years later, in 1979, by Tai [93] who considered the following set of elementary edit operations:

- *relabeling* of a node which changes the label of a node into a different one,
- *deletion* of a node which deletes a non-root node v making the children of v become children of $\mathsf{parent}(v)$, and
- *insertion* of a node v as a child of u making v the parent of a consecutive subsequence of the children of u.

A metric cost function is also assumed to assign each edit operation a non-negative real number. We refer to this type of edit distance as the *general tree edit distance*.

Tai also described a graphical specification of how a sequence of edit operations transforms one tree into another called *mapping*. Several authors used a restriction of the mapping between two trees to define a new type of edit distance between trees [94]–[97]. Another restricted variant of the general tree edit distance was considered, for example, by Shasha and Zhang [98] or Akmal and Jin [99]. They assumed that all operations come with the unit cost. Some authors considered a different set of operations, such as Lu [100], who used operations node split and node merge, or Klein [101], who considered edge-labeled trees and edit operations on edges.

The algorithmic nature of the tree edit distance problem and inexact tree pattern matching depends on the type of edit operations allowed and the type of trees considered [45]. For example, the tree edit distance problem is NP-hard for unordered labeled trees and the set of operations that consists of node relabeling, node deletion, and node insertion [102]. In the rest of this section, we narrow our attention to ordered labeled trees, which are the main focus of this dissertation thesis.

4.1.2.1 Dynamic programming approach

The majority of existing methods that solve the tree edit distance problem for ordered labeled trees are based on the dynamic programming approach [45]. The algorithms can be seen as an extension of the dynamic programming approach to the string edit distance problem that we described in Section 4.1.1.1.

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two ordered labeled trees. Let l_1 and l_2 be the number of leaves in $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. Also, let h_1 and h_2 be the depth of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. The first algorithm for computing the general tree edit distance between two ordered labeled trees was given by Tai [93]. Zhang and Shasha [69] gave a faster and more space-efficient dynamic programming algorithm that works in $\mathcal{O}(|\mathcal{T}_1| \cdot |\mathcal{T}_2| \cdot \mathsf{min}(l_1, h_1) \cdot \mathsf{min}(l_2, h_2))$ time and $\mathcal{O}(|\mathcal{T}_1| \cdot |\mathcal{T}_2|)$ space. Klein [103] improved the algorithm to run in $\mathcal{O}(|\mathcal{T}_1|^2 \cdot |\mathcal{T}_2| \cdot \log |\mathcal{T}_2|)$. Dulucq and Touzet [104] generalized the previous decomposition algorithms to a decomposition strategy framework and proved a lower bound time for any such strategy. In 2001, Chen [105] presented an algorithm that uses a reduction to a matrix multiplication problem. Chen's algorithm works in $\mathcal{O}\big(|\mathcal{T}_1| \cdot |\mathcal{T}_2| + l_1^2 \cdot |\mathcal{T}_2| + l_1^{2.5} \cdot l_2\big)$ time and $\mathcal{O}\big(\big(|\mathcal{T}_1| + l_1^2\big) \cdot \mathsf{min}(l_2, h_2) + |\mathcal{T}_2|\big)$ space. In 2010, Demaine et al. [106] presented $\mathcal{O}\big(|\mathcal{T}_1| \cdot |\mathcal{T}_2|^2 \cdot (1 + \log(|\mathcal{T}_1|/|\mathcal{T}_2|))\big)$ time algorithm working in $\mathcal{O}(|\mathcal{T}_1| \cdot |\mathcal{T}_2|)$ space that is based on dynamic programming and falls into the same decomposition strategy framework of Dulucq and Touzet [104], Klein [103], and Zhang and Shasha [69]. When both $\mathcal{T}_1$ and $\mathcal{T}_2$ has n nodes, the algorithm time complexity is $\mathcal{O}\big(n^3\big)$. According to a recent result [107], it is unlikely that there is a truly subcubic algorithm for the ordered tree edit distance problem.

Better time complexity can be achieved for restricted variants of the general tree edit distance [42], [95], [98], [99]. For example, Selkow [42] gave an algorithm running in $\mathcal{O}(|\mathcal{T}_1| \cdot |\mathcal{T}_2|)$ time and space to compute the 1-degree edit distance between two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$. The algorithm is again based on the dynamic programming approach in which the input trees are decomposed recursively into smaller subproblems. Since we use the 1-degree edit distance in Chapter 5 in which we introduce our automata-based approach to the inexact tree pattern matching under this distance, we present Selkow's algorithm in detail. Specifically, we provide a version of Selkow's algorithm where we assume that all operations have the unit cost.

▶ **Algorithm 4.7** (The dynamic programming approach to computing the simple 1-degree edit distance [42])**.** Let Σ be an alphabet. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two ordered labeled trees over Σ. Let $r_1 = \mathsf{root}(\mathcal{T}_1)$ and $r_2 = \mathsf{root}(\mathcal{T}_2)$. Given $\mathcal{T}_1$ and $\mathcal{T}_2$, the algorithm computes $d_s(\mathcal{T}_1, \mathcal{T}_2)$.

The principal internal data structure is a two-dimensional array $\boldsymbol{D}$ in which the dimensions have ranges 0 to $\mathsf{degree}(r_1)$ and 0 to $\mathsf{degree}(r_2)$, respectively. By $\boldsymbol{D}_{i,j}$, where $i \in \{0, \dots, \mathsf{degree}(r_1)\}$ and $j \in \{0, \dots \mathsf{degree}(r_2)\}$, the value at row i and column j is denoted. When the array is filled $\boldsymbol{D}_{i,j}$ contains $d_s(\mathcal{T}_1^i, \mathcal{T}_2^j)$, where $\mathcal{T}_1^i$ denotes the subtree of $\mathcal{T}_1$ that consists of the root of $\mathcal{T}_1$ and bottom-up subtrees $\mathcal{T}_1/v_1, \dots, \mathcal{T}_1/v_i$, where v_i is the i-th child of r_1 and $\mathcal{T}_2^j$ denotes the subtree of $\mathcal{T}_2$ that consists of the root of $\mathcal{T}_2$ and bottom-up subtrees $\mathcal{T}_2/u_1, \dots, \mathcal{T}_2/u_j$, where u_j is the j-th child of r_2. Thus, $\boldsymbol{D}_{\mathsf{degree}(r_1),\mathsf{degree}(r_2)}$ contains $d_s(\mathcal{T}_1, \mathcal{T}_2)$.

1. (Initialize array.)
   a. Set $\boldsymbol{D}_{0,0} \leftarrow \begin{cases} 0 & \text{if } \mathsf{label}(r_1) = \mathsf{label}(r_2), \\ 1 & \text{otherwise.} \end{cases}$
   b. For each $i \in \{1, \dots, \mathsf{degree}(r_1)\}$, set $\boldsymbol{D}_{i,0} \leftarrow \boldsymbol{D}_{i-1,0} + |\mathcal{T}_1/v_i|$, where v_i is the i-th child of r_1.
   c. For each $j \in \{1, \dots, \mathsf{degree}(r_2)\}$, set $\boldsymbol{D}_{0,j} \leftarrow \boldsymbol{D}_{0,j-1} + |\mathcal{T}_2/v_j|$, where v_j is the j-th child of r_2.
2. (Fill array.) For each $j \in \{1, \dots, \mathsf{degree}(r_2)\}$ and $i \in \{1, \dots, \mathsf{degree}(r_1)\}$:
   a. The cost of transforming $\mathcal{T}_1/v_i$ to $\mathcal{T}_2/v_j$ is $c_1 \leftarrow \boldsymbol{D}_{i-1,j-1} + d_s(\mathcal{T}_1/v_i, \mathcal{T}_2/v_j)$.
   b. The cost of deleting $\mathcal{T}_1/v_i$ from $\mathcal{T}_1$ is $c_2 \leftarrow \boldsymbol{D}_{i-1,j} + |\mathcal{T}_1/v_i|$.
   c. The cost of inserting $\mathcal{T}_2/v_j$ into $\mathcal{T}_1$ is $c_3 \leftarrow \boldsymbol{D}_{i,j-1} + |\mathcal{T}_2/v_j|$.
   d. Set $\boldsymbol{D}_{i,j} \leftarrow \mathsf{min}(c_1, c_2, c_3)$.
3. (Return.) Return $\boldsymbol{D}_{\mathsf{degree}(r_1),\mathsf{degree}(r_2)}$.

▶ **Example 4.8** (The dynamic programming approach to computing the simple 1-degree edit distance)**.** Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the ordered labeled trees illustrated in Figure 4.4. Algorithm 4.7 computes $d_s(\mathcal{T}_1, \mathcal{T}_2)$ as follows:

1. (Initialize array.) The two-dimensional array $\boldsymbol{D}$ in which the dimensions have ranges 0 to 3 and 0 to 2, respectively, is created. Since $\mathsf{label}(\mathsf{root}(\mathcal{T}_1)) = \mathsf{label}(\mathsf{root}(\mathcal{T}_2))$, value $\boldsymbol{D}_{0,0}$ is set to 0. Other initialization steps are as follows (see also Figure 4.5a):
   - $\boldsymbol{D}_{1,0} = \boldsymbol{D}_{0,0} + |\mathcal{T}_1/\mathtt{b}^2| = 0 + 1 = 1$,
   - $\boldsymbol{D}_{2,0} = \boldsymbol{D}_{1,0} + |\mathcal{T}_1/\mathtt{b}^3| = 1 + 1 = 2$,
   - $\boldsymbol{D}_{3,0} = \boldsymbol{D}_{2,0} + |\mathcal{T}_1/\mathtt{a}^4| = 2 + 2 = 4$,
   - $\boldsymbol{D}_{0,1} = \boldsymbol{D}_{0,0} + |\mathcal{T}_2/\mathtt{b}^2| = 0 + 1 = 1$, and
   - $\boldsymbol{D}_{0,2} = \boldsymbol{D}_{0,1} + |\mathcal{T}_2/\mathtt{b}^3| = 1 + 3 = 4$.

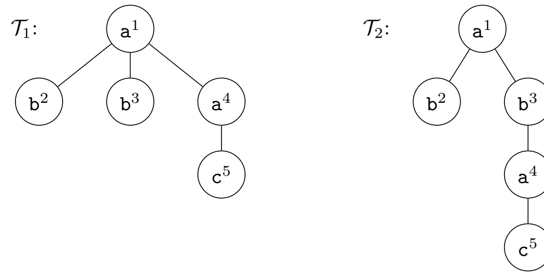
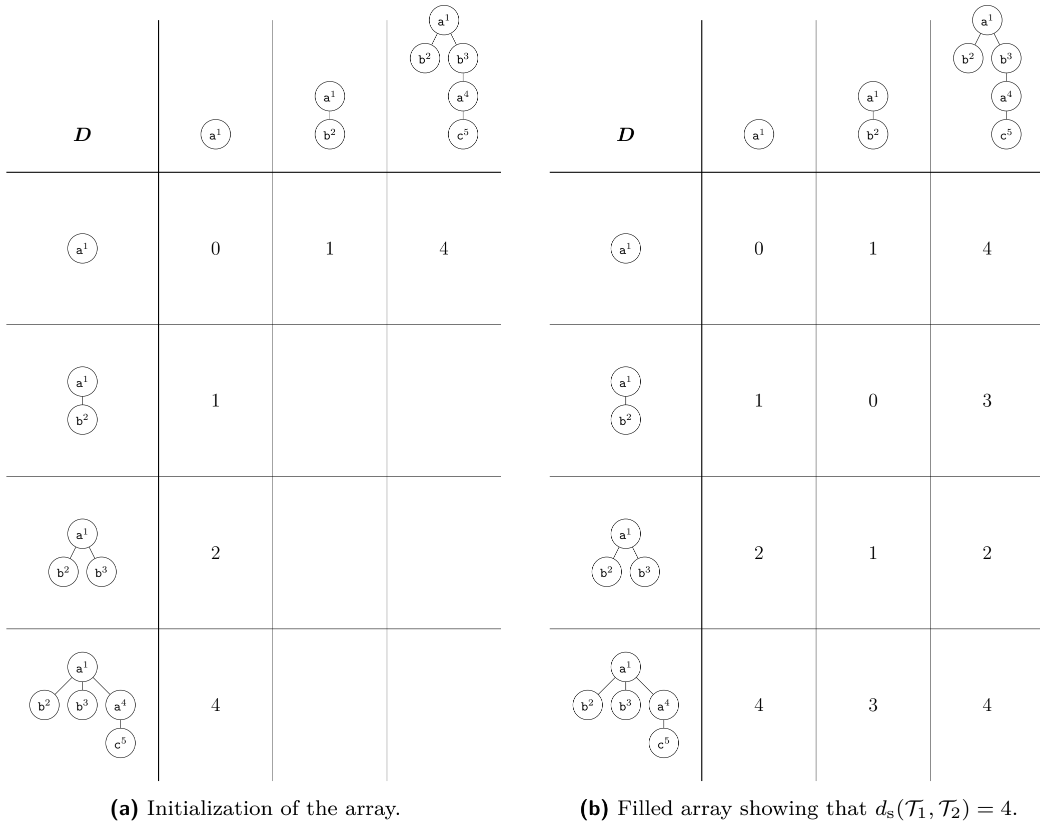


Figure 4.4 Pictorial representation of ordered labeled trees used in Example 4.8.



(a) Initialization of the array. **(b)** Filled array showing that $d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) = 4$.

Figure 4.5 Computation of the simple 1-degree edit distance between trees $\mathcal{T}_1$ and $\mathcal{T}_2$ illustrated in Figure 4.4. Let r_1 and r_2 be the root of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. By $\boldsymbol{D}_{i,j}$, where $i \in \{0, \ldots, \mathsf{degree}(r_1)\}$ and $j \in \{0, \ldots \mathsf{degree}(r_2)\}$, the value at row i and column j is denoted. When the array is filled $\boldsymbol{D}_{i,j}$ contains $d_{\mathrm{s}}(\mathcal{T}_1^i, \mathcal{T}_2^j)$, where $\mathcal{T}_1^i$ denotes the subtree of $\mathcal{T}_1$ that consists of the root of $\mathcal{T}_1$ and its bottom-up subtrees $\mathcal{T}_1/v_1, \ldots, \mathcal{T}_1/v_i$, where v_i is the i-th child of r_1 and $\mathcal{T}_2^j$ denotes the subtree of $\mathcal{T}_2$ that consists of the root of $\mathcal{T}_2$ and its bottom-up subtrees $\mathcal{T}_2/u_1, \ldots, \mathcal{T}_2/u_j$, where u_j is the j-th child of r_2. Therefore, $\boldsymbol{D}_{\mathsf{degree}(r_1),\mathsf{degree}(r_2)}$ contains $d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$.

2. (Fill array.) Other values are computed column-wise.

$$\boldsymbol{D}_{1,1} = \min \begin{cases} \boldsymbol{D}_{0,0} + d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{b}^2, \mathcal{T}_2/\mathsf{b}^2\big) & 0+0=0, \\ \boldsymbol{D}_{0,1} + \big|\mathcal{T}_1/\mathsf{b}^2\big| & 1+1=2, \\ \boldsymbol{D}_{1,0} + \big|\mathcal{T}_2/\mathsf{b}^2\big| & 1+1=2, \end{cases}$$

where $d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{b}^2, \mathcal{T}_2/\mathsf{b}^2\big)$ is computed recursively as illustrated in Figure 4.6a.

$$\boldsymbol{D}_{2,1} = \min \begin{cases} \boldsymbol{D}_{1,0} + d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{b}^3, \mathcal{T}_2/\mathsf{b}^2\big) & 1+0=1, \\ \boldsymbol{D}_{1,1} + \big|\mathcal{T}_1/\mathsf{b}^3\big| & 0+1=1, \\ \boldsymbol{D}_{2,0} + \big|\mathcal{T}_2/\mathsf{b}^2\big| & 2+1=3, \end{cases}$$

where $d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{b}^3, \mathcal{T}_2/\mathsf{b}^2\big)$ is computed recursively as illustrated in Figure 4.6c.

$$\boldsymbol{D}_{3,1} = \min \begin{cases} \boldsymbol{D}_{2,0} + d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{a}^4, \mathcal{T}_2/\mathsf{b}^2\big) & 2+2=4, \\ \boldsymbol{D}_{2,1} + \big|\mathcal{T}_1/\mathsf{a}^4\big| & 1+2=3, \\ \boldsymbol{D}_{3,0} + \big|\mathcal{T}_2/\mathsf{b}^2\big| & 4+1=5, \end{cases}$$

where $d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{a}^4, \mathcal{T}_2/\mathsf{b}^2\big)$ is computed recursively as illustrated in Figure 4.6g.

$$\boldsymbol{D}_{1,2} = \min \begin{cases} \boldsymbol{D}_{0,1} + d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{b}^2, \mathcal{T}_2/\mathsf{b}^3\big) & 1+2=3, \\ \boldsymbol{D}_{0,2} + \big|\mathcal{T}_1/\mathsf{b}^2\big| & 4+1=5, \\ \boldsymbol{D}_{1,1} + \big|\mathcal{T}_2/\mathsf{b}^3\big| & 0+3=3, \end{cases}$$

where $d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{b}^2, \mathcal{T}_2/\mathsf{b}^3\big)$ is computed recursively as illustrated in Figure 4.6d.

$$\boldsymbol{D}_{2,2} = \min \begin{cases} \boldsymbol{D}_{1,1} + d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{b}^3, \mathcal{T}_2/\mathsf{b}^3\big) & 0+2=2, \\ \boldsymbol{D}_{1,2} + \big|\mathcal{T}_1/\mathsf{b}^3\big| & 3+1=4, \\ \boldsymbol{D}_{2,1} + \big|\mathcal{T}_2/\mathsf{b}^3\big| & 1+3=4, \end{cases}$$

where $d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{b}^3, \mathcal{T}_2/\mathsf{b}^3\big)$ is computed recursively as illustrated in Figure 4.6e.

$$\boldsymbol{D}_{3,2} = \min \begin{cases} \boldsymbol{D}_{2,1} + d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{a}^4, \mathcal{T}_2/\mathsf{b}^3\big) & 1+3=4, \\ \boldsymbol{D}_{2,2} + \big|\mathcal{T}_1/\mathsf{a}^4\big| & 2+2=4, \\ \boldsymbol{D}_{3,1} + \big|\mathcal{T}_2/\mathsf{b}^3\big| & 3+3=6, \end{cases}$$

where $d_{\mathrm{s}}\big(\mathcal{T}_1/\mathsf{a}^4, \mathcal{T}_2/\mathsf{b}^3\big)$ is computed recursively as illustrated in Figure 4.6f and Figure 4.6b.

3. (Return.) $d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) = 4$. See Figure 4.5b.

Every tree edit distance can be used to define an inexact tree pattern matching problem [49], [52]. Moreover, existing algorithms for computing tree edit distance can be generalized to solve the inexact tree pattern matching problem [50], [69]. For example, Zhang and Shasha [69] gave a dynamic programming algorithm to compute the general tree edit distance and also showed how to adapt it with the same time complexity to inexact tree pattern matching. Later, Zhang [95] proposed a dynamic programming algorithm for computing the constrained edit distance and also discussed how the algorithm can be generalized to inexact tree matching with the same complexity. However, in most cases, the generalization of algorithms computing tree edit distance to algorithms solving inexact tree pattern matching is not explicitly discussed. For example, to the best of our knowledge, the extension of Selkow's algorithm to inexact tree pattern matching has not been addressed in the literature.

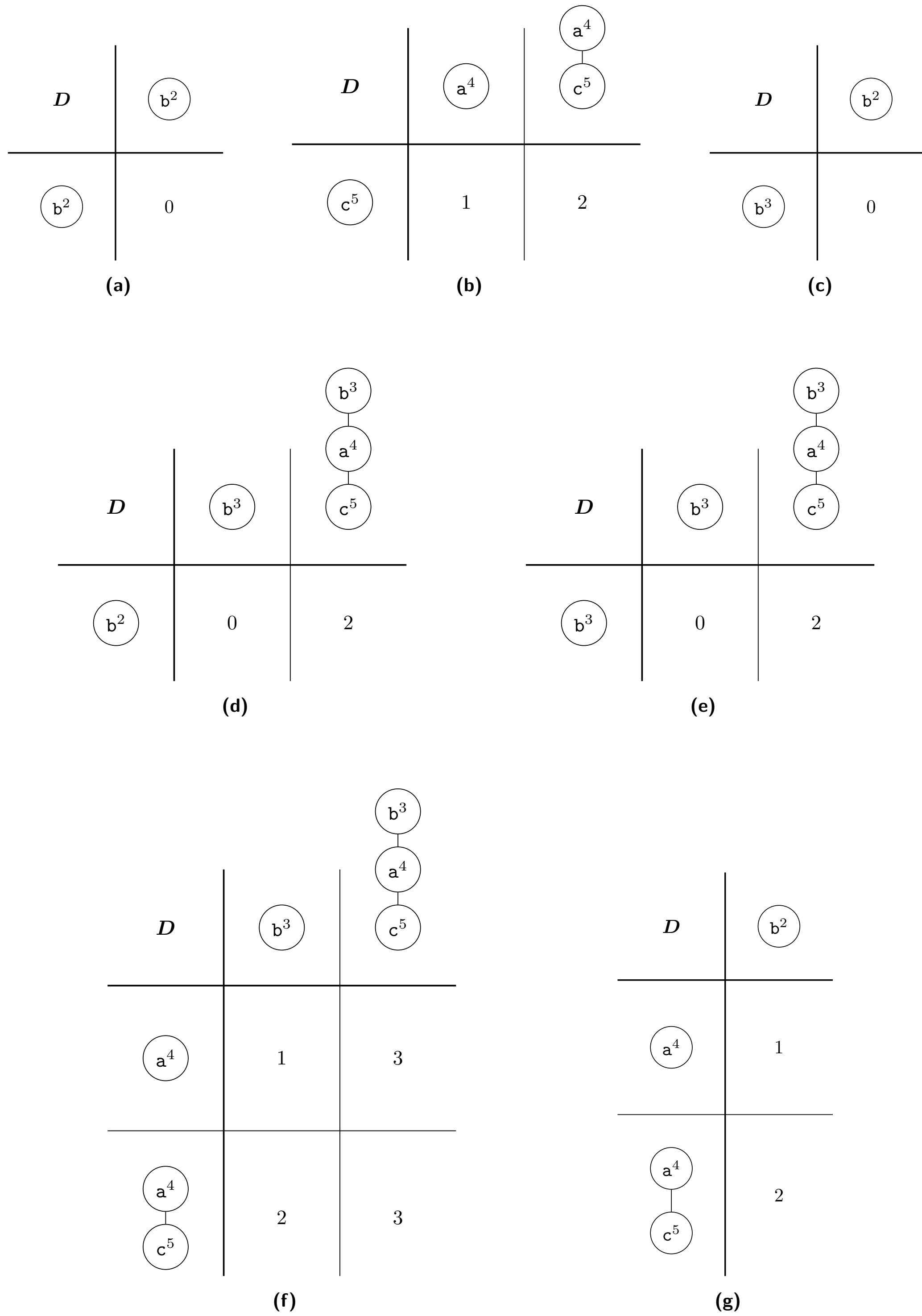


Figure 4.6 Recursive calls of Algorithm 4.7 for the computation of the simple 1-degree edit distance between trees $\mathcal{T}_1$ and $\mathcal{T}_2$ illustrated in Figure 4.4.

4.1.2.2 Automata approach

The automata approach to inexact pattern matching has been studied in the string domain, as discussed in Section 4.1.1.2. Moreover, it has been shown that there is a connection between this approach and the dynamic programming approach: the dynamic programming algorithm simulates the corresponding nondeterministic finite automaton. Therefore, it is natural to wonder whether the same approach can be applied to the inexact pattern matching in trees. However, to our knowledge, although the problem of inexact tree pattern matching has been addressed before [49]–[52], no existing solution is based on the automata-based approach.

In general, two automata-based approaches to inexact tree pattern matching problems may be explored. One way to process a tree structure is to use a computational model known as tree automaton [21], [22], which works directly on the tree structure and can be viewed as the generalization of a string automaton. As a textbook reference on finite tree automata, see the so-called "TATA book" by Comon et al. [21]. Another way is to encode trees as strings using linear notation and use string automata for their processing.

Research on the automata-based approach to tree pattern matching has mainly focused on exact matching. Below, we give an overview of problems addressed by arbology research using string automata. Since this dissertation thesis focuses on string automata, we do not discuss methods based on finite tree automata. For the finite tree automata approach to tree pattern matching, we recommend the work of Cleophas [22], who presents a systematic approach to solving the tree pattern matching problem using finite tree automata.

In general, string automata have been used in arbology to address online versions of three tree pattern matching problems: NFFOBE tree pattern matching problem (subtree matching), NFFFBE tree pattern matching problem (multiple subtree matching), and N(SW)FOBE tree pattern matching (tree template matching).

Flouri, Janoušek, and Melichar [23]–[25], [30], [38], [108] introduced a solution to the NFFOBE tree pattern matching problem that is based on a pushdown automaton called *subtree matching PDA*. They consider ordered ranked trees and represent them in prefix notation (or postfix notation [38]). Given an ordered ranked tree $\mathcal{P}$, the subtree matching PDA constructed for the prefix notation of $\mathcal{P}$ accepts all matches of $\mathcal{P}$ in any given input tree by final state. The subtree matching PDA is *input-driven*, which means that each pushdown operation is determined only by the input symbol and that the PDA can be transformed into an equivalent deterministic PDA. The deterministic subtree matching PDA has $|\mathcal{P}| + 1$ states, one pushdown symbol and $|\Sigma| \cdot (|\mathcal{P}| + 1)$ transitions. It finds all bottom-up subtrees in a given input that are isomorphic to $\mathcal{P}$ in time linear to the size of the input tree.

Moreover, as briefly discussed by Lahoda and Žďárek [36], a finite automaton can also be used for the NFFOBE tree pattern matching problem. Given an ordered labeled tree $\mathcal{P}$, a finite automaton for exact string pattern matching is constructed for the prefix bar notation of $\mathcal{P}$. This automaton reads the prefix bar notation of an input tree $\mathcal{T}$ and locates all bottom-up subtrees in $\mathcal{T}$ that match $\mathcal{P}$. The automaton can be constructed in time $\mathcal{O}(|\mathcal{P}|)$ and find all occurrences in time $\mathcal{O}(|\mathcal{T}|)$.

Flouri, Janoušek, and Melichar [23], [24], [30], [38], [109] generalized the subtree matching PDA to allow matching with multiple patterns. Given a set of ordered ranked trees $\mathcal{P}_1, \dots, \mathcal{P}_m$, the (multiple) subtree matching PDA constructed for the prefix notation of these trees accepts all matches of $\mathcal{P}_1, \dots, \mathcal{P}_m$ in any given input tree by final state. The (multiple) subtree matching PDA is input-driven, which means that it can be transformed into an equivalent deterministic PDA. The deterministic (multiple) subtree matching PDA finds all bottom-up subtrees in a given input that are isomorphic to some of the pattern trees in time linear to the size of the input tree and its space complexity is $\Theta\big(|\Sigma| \cdot \sum_{i=1}^{m} |\mathcal{P}_i|\big)$.

Flouri et al. [30], [110] considered the N(SW)FOBE tree pattern matching problem for ordered ranked trees in postfix notation and proposed a pushdown automaton called *tree template matching PDA*. The PDA belongs to the class of height-deterministic PDA [111], which means that it can be transformed into an equivalent deterministic PDA. Given a tree with subtree

wildcards $\mathcal{P}$, the tree template PDA constructed for the postfix notation of $\mathcal{P}$ finds all bottom-up subtrees in a given input that match $\mathcal{P}$ in time linear to the size of the input tree. In general, the space required for preprocessing the pattern tree with subtree wildcards is exponential to its size. However, the authors also discuss a specific class of pattern trees with subtree wildcards for which the space required is linear. Lahoda and Žďárek [36] studied a similarly stated problem using a pushdown automaton for trees represented in the prefix bar notation.

Several methods based on string automata have also been introduced for the offline variant of the tree pattern matching problem, in which the input tree is preprocessed instead of the pattern. We give an overview of these methods in Section 4.2.2.

4.2 Indexing

Standard approaches to pattern matching problems involve various methods for preprocessing a given pattern pattern so that its occurrences in the input data can be located fast. However, the search time depends on the size of the input data, which is inefficient when we search for many different patterns in the fixed input data. In this case, it is desirable to preprocess the input data, building an auxiliary data structure known as an index.

An index is an abstract data structure that provides efficient methods for answering queries related to the content of the input data, such as

- method for query membership that tests whether a given query has the empty or a nonempty set of answers in the data,
- method for query position that gives the position of the first or last occurrence of the query in the data,
- method for the number of occurrences of the query that gives the number of how many times the query occurs in the data, and
- methods for the list of all occurrences of the query that gives the list of all positions where the query occurs in the input data.

This chapter includes two sections. First, we describe some methods for string indexing. Then, we focus on the problem of indexing trees.

4.2.1 Strings

The theory of string indexing is well-researched [71], [112], [12, Chapter 3] and uses various data structures for efficient access to parts of a string, such as *suffix trees* [71, Section 5.2], [87, Part II], [113] or *suffix arrays* [114]. Since this dissertation thesis focuses on automata-based methods, we narrow our attention to automata indexes in this section. Specifically, we describe the automata indexes used in Chapter 6, in which we address the problem of indexing trees for linear XPath-inspired queries.

In general, indexes based on deterministic finite automata solve the membership problem for any given query in time linear to the length of the query (assuming that the time necessary to compute a transition from each state is constant). The issue could be the size of the index. However, as we discuss in this section, the remarkable result has been demonstrated that the size of a DFA indexing all substrings of a string is linear. The surprise is due to the maximum number of substrings occurring in a string which can be quadratic to the length of the string.

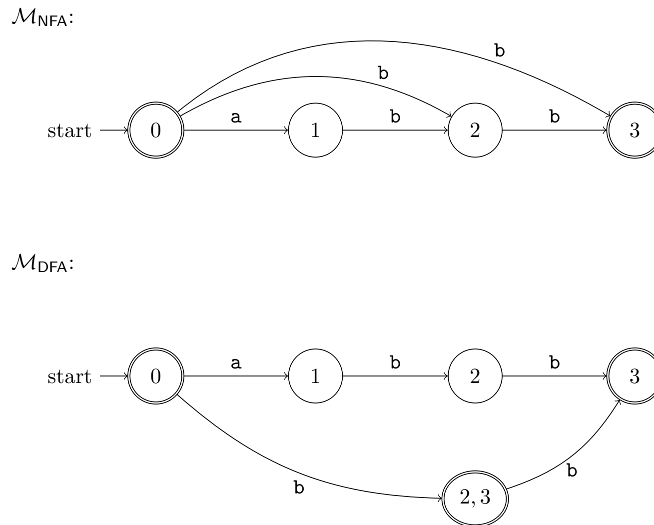


Figure 4.7 The nondeterministic suffix automaton for string $\boldsymbol{x} = \texttt{abb}$ constructed by Algorithm 4.9 and its (minimal) deterministic variant obtained by the subset construction.

4.2.1.1 Suffix automaton

In the literature, the notion of *suffix automaton* is often reserved for the minimal deterministic finite automaton that accepts the set of all suffixes of a string [10, Section 7], [115]. In this dissertation thesis, we use a more general definition: a suffix automaton is any finite automaton that accepts the set of all suffixes of a string.

A nondeterministic suffix automaton can be constructed to have a regular structure [12, Algorithm 3.10]. See Algorithm 4.9.

▶ **Algorithm 4.9** (Construction of a nondeterministic suffix automaton [12]). Let Σ be an alphabet. Let $\boldsymbol{x}$ be a nonempty string over Σ. Given $\boldsymbol{x}$, the algorithm constructs a nondeterministic finite automaton accepting the set of all suffixes of $\boldsymbol{x}$.

1. (Define the set of states.) Set $Q \leftarrow \{i : 0 \leq i \leq |\boldsymbol{x}|\}$.
2. (Define the start state.) Set $q_0 \leftarrow 0$.
3. (Define the set of final states.) Set $F \leftarrow \{0, |\boldsymbol{x}|\}$.
4. (Define transitions.) For each position i in $\boldsymbol{x}$:
   a. Set $\delta((i-1), \boldsymbol{x}_i) \leftarrow \{i\}$.
   b. Set $\delta(0, \boldsymbol{x}_i) \leftarrow \delta(0, \boldsymbol{x}_i) \cup \{i\}$.
5. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

We can turn the nondeterministic suffix automaton constructed by Algorithm 4.9 into a deterministic one using the subset construction; see Algorithm 2.6. Moreover, as we show in Theorem 4.10, the resulting DFA is minimal. The minimal deterministic suffix automaton has a linear number of states and transitions to the length of the input string [19], [20]. The minimal deterministic suffix automaton can also be constructed for a given input string online in time linear to the length of the string [19], [71, Theorem 5.29]. In Figure 4.7, we illustrate an example of the nondeterministic suffix automaton constructed by Algorithm 4.9 for string $\boldsymbol{x} = \texttt{abb}$ and its (minimal) deterministic variant obtained by the subset construction.

▶ **Theorem 4.10.** *Let Σ be an alphabet. Let $\boldsymbol{x}$ be a nonempty string over Σ. Let $\mathcal{M}$ be the nondeterministic suffix automaton for $\boldsymbol{x}$ constructed by Algorithm 4.9. Then, the deterministic suffix automaton $\mathcal{M}'$ obtained from $\mathcal{M}$ by the subset construction is minimal.*

Proof. We show that $\mathcal{M}'$ is minimal using some results given by Blumer et al. [19]. Thus, we begin by recalling a few notions they defined.

Let $\boldsymbol{w} \in \Sigma^+$. Then, $\mathsf{endSet}_{\boldsymbol{x}}(\boldsymbol{w}) = \{i : \boldsymbol{w} = \boldsymbol{x}_{i-|\boldsymbol{w}|+1\ldots i}\}$. Moreover, $\mathsf{endSet}_{\boldsymbol{x}}(\varepsilon) = \{0, 1, \ldots, |\boldsymbol{x}|\}$. It is said that $\boldsymbol{y}, \boldsymbol{z} \in \Sigma^*$ are *end-equivalent* on $\boldsymbol{x}$, denoted by $\boldsymbol{y} \equiv_{\boldsymbol{x}} \boldsymbol{z}$, if $\mathsf{endSet}_{\boldsymbol{x}}(\boldsymbol{y}) = \mathsf{endSet}_{\boldsymbol{x}}(\boldsymbol{z})$. The equivalence class of $\boldsymbol{w}$ with respect to $\equiv_{\boldsymbol{x}}$ is denoted by $[\boldsymbol{w}]_{\boldsymbol{x}}$. Using these notions, Blumer et al. defined a deterministic finite automaton $\mathcal{S}_{\boldsymbol{x}} = (Q, \Sigma, \delta_{\mathcal{S}}, q_0, F)$ for $\boldsymbol{x}$ as follows:

- $Q = \{[\boldsymbol{w}]_{\boldsymbol{x}} : \boldsymbol{w}$ is a substring of $\boldsymbol{x}\}$,
- $q_0 = [\varepsilon]_{\boldsymbol{x}}$,
- $F = \{[\boldsymbol{w}]_{\boldsymbol{x}} : \boldsymbol{w}$ is a suffix of $\boldsymbol{x}\}$, and
- $\delta_{\mathcal{S}}([\boldsymbol{w}]_{\boldsymbol{x}}, a) = [\boldsymbol{w}a]_{\boldsymbol{x}}$, where $\boldsymbol{w}$ and $\boldsymbol{w}a$ are substrings of $\boldsymbol{x}$ and $a \in \Sigma$.

For $\mathcal{S}_{\boldsymbol{x}}$, Blumer et al. proved that it is the minimal deterministic suffix automaton for $\boldsymbol{x}$ [19, Proposition 1.3]. To prove that $\mathcal{M}'$ is minimal, we show that it is *isomorphic*[1] to $\mathcal{S}_{\boldsymbol{x}}$.

It is easy to see that $\mathcal{M}$ and $\mathcal{S}_{\boldsymbol{x}}$ can read a string (without failing due to a nonexisting transition) if and only if the string is a substring of $\boldsymbol{x}$. Let $\boldsymbol{w}$ be a nonempty substring of $\boldsymbol{x}$. For $\mathcal{S}_{\boldsymbol{x}}$, we clearly get that $\hat{\delta}_{\mathcal{S}}([\varepsilon]_{\boldsymbol{x}}, \boldsymbol{w}) = [\boldsymbol{w}]_{\boldsymbol{x}}$. For $\mathcal{M}$, we show that $\hat{\delta}(0, \boldsymbol{w}) = \{i : \boldsymbol{x}_{i-|\boldsymbol{w}|+1\ldots i}\}$, where δ is the transition function of $\mathcal{M}$. We use induction on the length of $\boldsymbol{w}$:

- Assume $|\boldsymbol{w}| = 1$. From Step 4 of Algorithm 4.9, we get that $\delta(0, a) = \{i : \boldsymbol{x}_i = a\}$. Thus, the claim holds for $|\boldsymbol{w}| = 1$.
- Assume $|\boldsymbol{w}| \geq 2$ and that the claim holds for all shorter substrings. Clearly, $\boldsymbol{w} = \boldsymbol{z}a$, where $\boldsymbol{z} \in \Sigma^+$ and $a \in \Sigma$. By the induction hypothesis, we have that $\hat{\delta}(0, \boldsymbol{z}) = \{i : \boldsymbol{x}_{i-|\boldsymbol{z}|+1\ldots i}\}$. Since there is a transition in $\mathcal{M}$ labeled by a from state j to state $j+1$ only if $\boldsymbol{x}_{j+1} = a$ (see Step 4a of Algorithm 4.9, we get that $\hat{\delta}(0, \boldsymbol{z}a) = \{i : \boldsymbol{x}_{i-|\boldsymbol{z}a|+1\ldots i}\}$. Thus, the claim holds.

Since the subset construction creates only states such that each corresponds to a subset of NFA states that are active after reading some input string, it follows that the set of states of $\mathcal{M}'$ consists of state $\{0\}$ and states in the form of $\{i : \boldsymbol{x}_{i-|\boldsymbol{w}|+1\ldots i}\}$. Clearly, state $\{0\}$ corresponds to state $[\varepsilon]_{\boldsymbol{x}}$ and each state $\{i : \boldsymbol{x}_{i-|\boldsymbol{w}|+1\ldots i}\}$ corresponds to state $[\boldsymbol{w}]_{\boldsymbol{x}}$. Furthermore, the state in $\mathcal{M}'$ is final if it contains $|\boldsymbol{x}|$ in its subset. This corresponds to the condition that the state in $\mathcal{S}_{\boldsymbol{x}}$ is final if it contains a suffix of $\boldsymbol{x}$ in its equivalence class. Therefore, $\mathcal{M}'$ is isomorphic to $\mathcal{S}_{\boldsymbol{x}}$, and since $\mathcal{S}_{\boldsymbol{x}}$ is minimal, $\mathcal{M}'$ is minimal too. ◀

▶ **Theorem 4.11** (Number of states of the minimal deterministic suffix automaton [19], [71]). *Let Σ be an alphabet. Let $\boldsymbol{x}$ be a string over Σ such that $|\boldsymbol{x}| \geq 2$. The number of states of the minimal deterministic suffix automaton for $\boldsymbol{x}$ is at least $|\boldsymbol{x}| + 1$ and at most $2|\boldsymbol{x}| - 1$.*

▶ **Theorem 4.12** (Number of transitions of the minimal deterministic suffix automaton [19], [71]). *Let Σ be an alphabet. Let $\boldsymbol{x}$ be a string over Σ such that $|\boldsymbol{x}| \geq 3$. The number of transitions of the minimal deterministic suffix automaton for $\boldsymbol{x}$ is at least $|\boldsymbol{x}|$ and at most $3|\boldsymbol{x}| - 4$.*

The deterministic suffix automaton can also be constructed for a given string so that its underlying graph structure forms a tree and where final states have a one-to-one correspondence with suffixes. This can be done, for example, by successively adding the suffixes into the tree,

[1] Two deterministic finite automata are isomorphic if they are the same up to the renaming of states. For precise definition, see, for example, Kozen [54, Page 89].

starting from the longest to the empty string. The resulting automaton is called the *suffix trie* [71, Section 5.1]. The main disadvantage of the suffix trie is that its number of states can be quadratic to the length of the string.

An automaton with a linear number of states can be obtained from the suffix trie by application of the standard minimization algorithm or by the process called *compaction* where non-final states with only one outgoing transition are deleted. Strings then label the transitions instead of symbols. This data structure is called *suffix tree* [113], [71, Section 5.2], [87, Part II]. Assuming that the string is stored in memory together with the suffix tree, its transition labels can be encoded so that the total size of the suffix tree is linear to the length of the string [71, Proposition 5.4].

Compaction can also be applied to the minimal deterministic suffix automaton to obtain an automaton with a smaller number of states and transitions [116], [117], [71, Page 202]. The compact minimal deterministic suffix automaton for a string $\boldsymbol{x}$ of length greater than two has at most $|\boldsymbol{x}| + 1$ states and at most $2|\boldsymbol{x}| - 2$ transitions.

4.2.1.2 Factor automaton

A *factor automaton* is a finite automaton that accepts the set of all substrings (factors) of a given string. We note that our definition is again more general since we do not reserve this notion for the minimal DFA, as it is often used in the literature [10, Section 7.6], [115].

Since every substring is a prefix of some suffix, the factor automaton can be constructed as a suffix automaton with all states final [12, Section 3.3]. See Algorithm 4.13. In other words, the suffix automaton gives us not only direct access to suffixes of a string but also its substrings. In the literature, both suffix automaton and factor automaton are also known under the name of *Directed Acyclic Word Graph* (DAWG) [19], [73], [11, Chapter 6].

▶ **Algorithm 4.13** (Construction of a nondeterministic factor automaton [12])**.** Let Σ be an alphabet. Let $\boldsymbol{x}$ be a nonempty string over Σ. Given $\boldsymbol{x}$, the algorithm constructs a nondeterministic finite automaton accepting the set of all substrings of $\boldsymbol{x}$.

1. (Define the set of states.) Set $Q \leftarrow \{i : 0 \leq i \leq |\boldsymbol{x}|\}$.
2. (Define the start state.) Set $q_0 \leftarrow 0$.
3. (Define the set of final states.) Set $F \leftarrow Q$.
4. (Define transitions.) For each position i in $\boldsymbol{x}$:
   a. Set $\delta\left((i-1), \boldsymbol{x}_i\right) \leftarrow \{i\}$.
   b. Set $\delta(0, \boldsymbol{x}_i) \leftarrow \delta(0, \boldsymbol{x}_i) \cup \{i\}$.
5. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

We can turn the nondeterministic factor automaton constructed by Algorithm 4.13 into an equivalent deterministic one using the subset construction. However, the resulting DFA does not have to be minimal. In Figure 4.8, we illustrate an example of the nondeterministic factor automaton constructed by Algorithm 4.13 for string $\boldsymbol{x} = \texttt{abb}$ and its deterministic variant obtained by the subset construction. Note that states $\{2\}$ and $\{2, 3\}$ are equivalent. Using the standard minimization algorithm, we can obtain the minimal deterministic factor automaton.

Although the maximum number of substrings occurring in a string can be quadratic to the length of the string, the number of states and transitions of the minimal deterministic factor automaton is linear to the length of the input string [19], [20]. Moreover, the minimal deterministic factor automaton can also be constructed directly online in time linear to the length of the input string [19], [20].

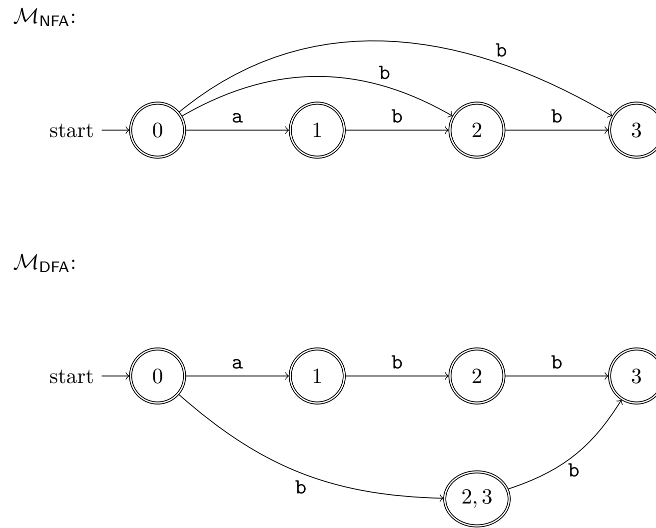


Figure 4.8 The nondeterministic factor automaton for string $\boldsymbol{x} = \mathtt{abb}$ constructed by Algorithm 4.13 and its deterministic variant obtained by the subset construction.

▶ **Theorem 4.14** (Number of states of the minimal deterministic factor automaton [19]). *Let Σ be an alphabet. Let $\boldsymbol{x}$ be a string over Σ such that $|\boldsymbol{x}| \geq 3$. The number of states of the minimal deterministic factor automaton for $\boldsymbol{x}$ is at least $|\boldsymbol{x}| + 1$ and at most $2|\boldsymbol{x}| - 2$.*

▶ **Theorem 4.15** (Number of transitions of the minimal deterministic factor automaton [19]). *Let Σ be an alphabet. Let $\boldsymbol{x}$ be a string over Σ such that $|\boldsymbol{x}| \geq 3$. The number of transitions of the minimal deterministic factor automaton for $\boldsymbol{x}$ is at least $|\boldsymbol{x}|$ and at most $3|\boldsymbol{x}| - 4$.*

4.2.1.3 Prefix automaton

A *prefix automaton* is a finite automaton that accepts the set of all prefixes of a given string. Since every prefix is a substring, the prefix automaton can be constructed using the simplified version of the algorithm that builds the factor automaton [12, Section 3.1]. See Algorithm 4.16. The resulting automaton is deterministic and minimal. We illustrate an example of the prefix automaton constructed by this algorithm for string $\boldsymbol{x} = \mathtt{abb}$ in Figure 4.9. The minimal deterministic prefix automaton for string $\boldsymbol{x}$ has clearly $|\boldsymbol{x}| + 1$ states and $|\boldsymbol{x}|$ transitions.

▶ **Algorithm 4.16** (Construction of a minimal deterministic prefix automaton [12]). Let Σ be an alphabet. Let $\boldsymbol{x}$ be a nonempty string over Σ. Given $\boldsymbol{x}$, the algorithm constructs the minimal deterministic finite automaton accepting the set of all prefixes of $\boldsymbol{x}$.

1. (Define the set of states.) Set $Q \leftarrow \{i : 0 \leq i \leq |\boldsymbol{x}|\}$.
2. (Define the start state.) Set $q_0 \leftarrow 0$.
3. (Define the set of final states.) Set $F \leftarrow Q$.
4. (Define transitions.) For each position i in $\boldsymbol{x}$, set $\delta\left((i-1), \boldsymbol{x}_i\right) \leftarrow \{i\}$.
5. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

$\mathcal{M}_{\mathsf{DFA}}$:

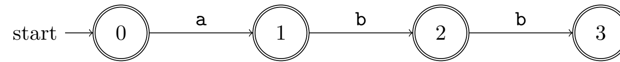


Figure 4.9 The (minimal deterministic) prefix automaton for string $\boldsymbol{x} = \mathtt{abb}$ constructed by Algorithm 4.16.

4.2.1.4 Subsequence automaton

A *subsequence automaton* is a finite automaton that accepts the set of all subsequences of a given string. We note that our definition is again more general since we do not reserve this notion for the minimal DFA. The (minimal) automaton solving the problem of subsequences is also in the literature referred to as the *Directed Acyclic Subsequence Graph* (DASG) [118], [119].

A nondeterministic subsequence automaton can be constructed to have a regular structure [12, Section 3.5]. See Algorithm 4.17. The process can be seen as an extension of the algorithm that builds the nondeterministic suffix (or factor) automaton.

▶ **Algorithm 4.17** (Construction of a nondeterministic subsequence automaton [12])**.** Let Σ be an alphabet. Let $\boldsymbol{x}$ be a nonempty string over Σ. Given $\boldsymbol{x}$, the algorithm constructs a nondeterministic finite automaton accepting the set of all subsequences of $\boldsymbol{x}$.

1. (Define the set of states.) Set $Q \leftarrow \{i : 0 \leq i \leq |\boldsymbol{x}|\}$.
2. (Define the start state.) Set $q_0 \leftarrow 0$.
3. (Define the set of final states.) Set $F \leftarrow Q$.
4. (Define transitions.) For each position i in $\boldsymbol{x}$ and for each $j \in \{0, \ldots, i-1\}$, set $\delta(j, \boldsymbol{x}_i) \leftarrow \delta(j, \boldsymbol{x}_i) \cup \{i\}$.
5. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

We can turn the nondeterministic subsequence automaton constructed by Algorithm 4.17 into an equivalent deterministic one using the subset construction. As we show in Theorem 4.18, the resulting DFA is minimal. In Figure 4.10, we illustrate an example of the nondeterministic subsequence automaton constructed by Algorithm 4.17 for string $\boldsymbol{x} = \mathtt{abbaccabc}$ and its (minimal) deterministic variant obtained by the subset construction.

▶ **Theorem 4.18.** *Let Σ be an alphabet. Let $\boldsymbol{x}$ be a nonempty string over Σ. Let $\mathcal{M}$ be the nondeterministic subsequence automaton for $\boldsymbol{x}$ constructed by Algorithm 4.17. Then, the deterministic subsequence automaton obtained from $\mathcal{M}$ by the subset construction has $|\boldsymbol{x}| + 1$ states and thus is minimal.*

Proof. Let Q be the set of states of $\mathcal{M}$. First, the subset construction creates the start state of the DFA, which is the state $\{0\}$. Then, for each $a \in \Sigma$, the subset construction creates a state q_a such as $\delta(\{0\}, a) = q_a$. It is easy to check that $q_a = \{i : \boldsymbol{x}_i = a\}$ for each $a \in \Sigma$ and that $\bigcup_{a \in \Sigma} q_a = Q \setminus \{0\}$. Moreover, for each $a, b \in \Sigma$ such that $a \neq b$, we get that the sets q_a and q_b are disjoint. Since every state in $\mathcal{M}$ has all the incoming transitions labeled by the same symbol (homogeneous automaton), it follows that all other states created during the subset construction are subsets of q_a for some $a \in \Sigma$. Moreover, from the structure of $\mathcal{M}$, it follows that if the subset construction creates a state (a subset) which contains state i of $\mathcal{M}$, then the state (subset) must also contain every state $j > i$ of $\mathcal{M}$, where $\boldsymbol{x}_i = \boldsymbol{x}_j$. Therefore, the maximum number of states (subsets) that can be created during the subset construction is as follows: 1 (the start state) +

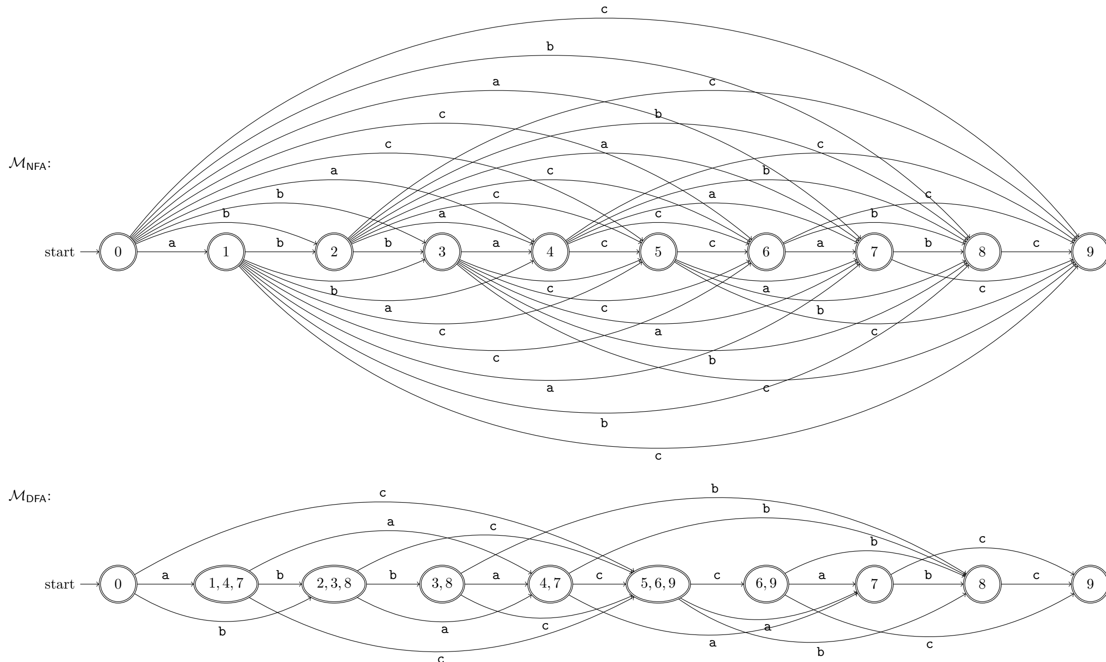


Figure 4.10 The nondeterministic subsequence automaton for string $\boldsymbol{x} = \texttt{abbaccabc}$ constructed by Algorithm 4.17 and its (minimal) deterministic variant obtained by the subset construction.

$|\Sigma|$ (the target states of transitions leading from the start state) + $\sum_{a \in \Sigma}(|q_a| - 1)$ (the maximum number of other states that can be created). This results in $|Q| = |\boldsymbol{x}| + 1$ states. The minimum number of states of the DFA is also $|\boldsymbol{x}| + 1$ since the automaton has to accept the complete string (the longest subsequence), and this requires $|\boldsymbol{x}| + 1$ states in the automaton accepting a finite language. Thus, the DFA has exactly $|\boldsymbol{x}| + 1$ states, and it is minimal. ◀

The (state) minimal deterministic subsequence automaton for a string $\boldsymbol{x}$ has $|\boldsymbol{x}| + 1$ states and $\mathcal{O}(|\Sigma| \cdot |\boldsymbol{x}|)$ transitions [118]. This is a remarkable result since the maximum number of subsequences occurring in a string can be exponential to the length of the string. However, as discussed by Baeza-Yates [118], the minimal state automaton is in general not the minimal space automaton because of the large number of transitions. To reduce the number of transitions, Baeza-Yates introduced an encoding technique that results in $\mathcal{O}(|\boldsymbol{x}| \cdot \log |\Sigma|)$ space representation of the subsequence automaton.

4.2.1.5 Patterns with gaps

In this section, we explore the previous results that address the problem of string indexing for *patterns with gaps* (or *gapped patterns*)[2]. This is because the problem of indexing trees for linear gapped pattern trees, the focus of Chapter 6, can be seen as a variant of this problem. Specifically, the problem that arises in Chapter 6 can be seen as the problem of indexing a string (or a set of strings) for prefixes with unbounded gaps. Formally, given a string $\boldsymbol{x}$ over alphabet Σ, the index should be able to recognize all gapped patterns that match a prefix of $\boldsymbol{x}$. By a gapped pattern, we denote a string over $\Sigma \cup \{\wr\}$ of the form $g_1 p_1 g_2 p_2 \ldots g_n p_n$, where g_i is either the empty string or $\wr$ and $p_i \in \Sigma$. During matching, the symbol $\wr$ is interpreted to match an arbitrary substring of

[2] We assume that the text does not contain gaps.

$\boldsymbol{x}$ of any length (even zero). For example, given a text $\boldsymbol{w}$ = `abbaabcbba`, there is one occurrence of the gapped pattern `abb≀c` (end position 7), and there are three occurrences of the gapped pattern `≀aa≀b` (end positions 6, 8, and 9).

In the literature, a gap in a pattern is usually interpreted as follows:

- A gap is a special symbol that matches any symbol in the alphabet. This type of gap is also called *don't care symbol* or *wildcard* [120]–[123]. Moreover, there is also the notion of *optional wildcard*, which is a special symbol that matches any symbol as well as no symbol [124].
- A gap comes with a lower and upper bound, which specifies the minimum and maximum length of a substring it can match. Patterns with this type of gaps are called *variable-length-gapped patterns* [125]–[127]. Note that a don't care symbol is equivalent to the gap with bounds equal to one.
- A gap is *unbounded* which means that it matches a substring of any length (even zero). The unbounded gap corresponds to the gap with the lower bound equal to zero and the upper bound equal to the length of text in which the pattern is being searched.

Usually, the problem of matching gapped patterns is about finding substrings in the text that match a given gapped pattern. Therefore, gaps are usually allowed to be present only inside the pattern and not at its beginning.

There has been extensive research on the non-indexing variant of the matching pattern with gaps [125], [128], [129]. As for the indexing variant, Rahman and Iliopoulos [121], Lam et al. [124] and Bille et al. [130] have, for example, proposed an index for matching patterns with don't care symbols. The indexing variant of the variable-length-gap pattern matching has been considered, for example, by Lewenstein [126], Bader et al. [131], and Cáceres et al. [127]. Solutions have also been proposed for the case of one gap where patterns are called *gapped-factors*. A gapped factor is a concatenation of a factor, a gap of a given length, and another factor [122], [132]. However, to our knowledge, no one has studied an automata-based approach to the problem of string indexing for patterns (prefixes) with unbounded gaps. The most relevant existing work we have found is the automata-based index for the problem of indexing the common *motifs*[3] with gaps in a set of strings proposed by Antoniou et al. [13]. The index uses factor automata constructed for each input string and an automaton that accepts the union of languages accepted by these automata.

4.2.2 Trees

In this section, we give an overview of existing methods for the problem of tree indexing that are related to the focus of this dissertation thesis. First, we describe previous automata-based indexes for trees that have been a result of arbology research. Then, we discuss approaches to tree indexing used in the XML domain.

In general, string automata have been used in arbology to address offline (indexing) versions of four tree pattern matching problems: NFFOBE tree pattern matching problem (subtree matching), NFFFBE tree pattern matching problem (multiple subtree matching), N(SW)FOBE tree pattern matching (tree template matching), and N(SV)FOBE tree pattern matching (non-linear tree template matching). These methods are based on pushdown automata accepting by empty pushdown store, and their construction is inspired by the existing approach to string indexing discussed in the previous section.

For the offline variant of the NFFOBE tree pattern matching problem, Janoušek [133] introduced a pushdown automaton called *subtree PDA*, which is an index for all bottom-up subtrees of an ordered ranked tree in prefix notation. The subtree PDA has both nondeterministic and deterministic variant. In its underlying graph structure, the nondeterministic subtree PDA

[3] Given a set of input strings, the problem of finding common motifs is the problem of finding similar substrings that are shared by all (or some) of the input strings.

is analogous to the nondeterministic suffix automaton constructed by Algorithm 4.9. However, due to the pushdown operations, some of the states and transitions that are present in the deterministic suffix automaton obtained by the subset construction need not be present in the deterministic subtree PDA. This is because the corresponding pushdown operations cannot be performed; therefore, such states are unreachable. The total size of the deterministic subtree PDA is not greater than $2|\mathcal{T}| + 1$ states and $3|\mathcal{T}|$ transitions, where $\mathcal{T}$ is the indexed tree. Assuming that the time necessary to compute a transition from each state is constant, the search phase of all occurrences of a subtree is performed in time linear to the size of a given query and does not depend on the size of the indexed tree. The subtree PDA for ordered unranked trees in the prefix bar notation was presented by Flouri et al. [27], who also showed how to use this automaton to find all subtree repeats.

Inspired by the oracle[4] modification [72] to the factor automaton used in stringology, Plicka, Janoušek, and Melichar [28] modified the subtree PDA and introduced the subtree oracle PDA for ordered labeled trees (both ranked and unranked). The subtree oracle PDA has $n + 1$ states, where n is the length of the corresponding linear notation of the indexed tree. The automaton can also accept strings that do not represent a bottom-up subtree in the indexed tree. However, as discussed by the authors, the number of such false positive matches is smaller than in the case of the (string) factor oracle automaton.

Later, Poliak et al. [34], [35] made use of regularities that often occur in trees and proposed a deterministic PDA called *tree compression automaton*. This automaton is built for a set of ordered unranked trees in prefix bar notation and accepts all bottom-up subtrees of given trees by empty pushdown store. In the worst case, the construction algorithm creates a tree compression automaton whose size is linear to the size of the indexed tree(s). In the best case, the size of the tree compression automaton is logarithmic. Thus, the tree compression automaton can also be used for compressing the input tree.

As an extension of subtree PDA, Melichar, Janoušek, Flouri, and Trávníček [23], [24], [30], [31] proposed the *tree pattern PDA* which is an index for all trees with subtree wildcards (we recall Definition 3.7) for which there exists a matching bottom-up subtree in the input tree. As discussed by the authors, searching for tree patterns when trees are encoded using a linear tree notation can be seen as a variant of string matching with variable-length gaps that we briefly discussed in Section 4.2.1.5. The subtree wildcard represents a gap in the string representation of the pattern tree. However, such gaps are of unknown size, and a condition is placed on the matched string to be a valid linear representation of a tree.

The nondeterministic tree pattern PDA is input-driven and thus can be transformed to an equivalent deterministic PDA. Assuming that the time necessary to compute a transition from each state is constant, the deterministic tree pattern PDA finds all occurrences of a tree pattern in time linear to the size of the pattern and not depending on the size of the indexed tree. However, the total size of the deterministic tree pattern PDA can be exponential to the size of the indexed tree. Poliak [35] showed that there exists a tree $\mathcal{T}$ such that the minimal deterministic tree pattern PDA for $\mathcal{T}$ is of size $\mathcal{O}(2^{|\mathcal{T}|/4})$.

Although the size of the deterministic tree pattern PDA is exponential, the size of its nondeterministic variant is linear. Janoušek et al. [32] used this property and proposed a simulation algorithm with linear space complexity consisting of the compact suffix automaton and an auxiliary structure called the *subtree jump table*. The subtree jump table is a data structure that provides information on the start and end positions of the bottom-up subtrees. In Section 4.2.2.2, we present the precise definition, examples, and properties of the subtree jump table since we use it for our automata-based methods introduced in Chapter 5.

As an extension of tree pattern PDA, Trávníček et al. [29], [31] proposed the *nonlinear tree pattern PDA*, which is an index for all trees with subtree variables (we recall Definition 3.11) for which there exists a matching bottom-up subtree in the input tree. The automaton is constructed for the prefix notation of an ordered ranked tree, and since it is input-driven, it can be transformed

[4] Oracle matching can report some false positives as matches.

into an equivalent deterministic PDA. However, the exact space complexity of the deterministic nonlinear tree pattern PDA is an open question. Later, Trávníček [33] extended the linear index presented by Janoušek et al. [32] to provide a linear space tree index for all trees with subtree variables.

4.2.2.1 Linear patterns

To date, arbology research has not addressed the problem of indexing trees for linear patterns; that is, queries that can be represented as linear trees. However, this problem has attracted considerable attention from database researchers [134]–[138] since linear patterns naturally arise, for example, in querying XML using the XPath query language. For a survey on indexing XML documents, see, for example, Luk et al. [139], Catania et al. [140], or Mohammad [141].

Catania et al. [140] classified XML document indexes based on the types of supported queries and the indexing strategies. The authors considered three criteria for classifying XML queries: a tree structure, starting node, and node types. According to its tree structure, XML queries can be either *simple path* or *branching*. In this dissertation thesis, we call these types of queries linear patterns and non-linear patterns, respectively. The starting node criterion distinguishes between queries whose matching starts from the root (*total matching*) or any node (*partial matching*). We note that total matching corresponds to the top-down subtree way of matching, and partial matching can be interpreted as the subtree way of matching; see Section 3.1.5. The last criterion for the XML query classification considered by the authors, called node types, concerns the element content, not the graph structure. The proposed classification of XML document indexes distinguishes between three techniques called *summary* [134], [135], [137], [142], [143], *structural join* [136], [144], and *sequence-based* indexes [145], [146].

The most relevant to the focus of this dissertation thesis are summary indexes that provide efficient support for simple path queries (linear patterns). Branching queries (nonlinear patterns) are usually decomposed into simple path queries, whose results are then merged. Goldman and Widom [134] proposed a summary index called *DataGuides*. The theory behind DataGuides is that its construction over a database can be seen as a transformation of an NFA to an equivalent DFA, which in the case of tree-like database takes linear time [147]. Milo et al. [135] proposed an index called *1-index*, which can be seen as a nondeterministic version of DataGuide.

String automata found their application in XML not only as summary indexes. Segoufin and Vianu [148], for example, considered finite automata in the context of XML validation. String automata have also been used for pattern matching and filtering [149] in which the query (or a set of queries) is preprocessed. For example, Green et al. [150], [151] converted a set of XPath queries into a DFA that was then used for query evaluation of the input XML stream. Later, Kumar et al. [152] proposed visibly pushdown automata as the model suitable for processing streaming XML. Apart from string automata, tree automata also found their application in XML [153]–[155].

4.2.2.2 Subtree jump table

In this section, we focus on the data structure proposed by Janoušek et al. [32] called the subtree jump table. Specifically, we focus on the variant of the subtree jump table proposed by Trávníček [33, Section 5.2.2] that contains start and end positions for all the bottom-up subtrees of an ordered labeled tree in the prefix bar notation. See Definition 4.19 and Figure 4.11.

▶ **Definition 4.19** (Subtree jump table for prefix bar notation [33])**.** Let Σ be an alphabet. Let $\boldsymbol{x} \in L(\Sigma, \uparrow)$. The *subtree jump table* for $\boldsymbol{x}$ is an array $\boldsymbol{S}^{\boldsymbol{x}}$ in which the dimension has ranges 1 to $|\boldsymbol{x}|$. Values in $\boldsymbol{S}^{\boldsymbol{x}}$ are defined as follows: $\boldsymbol{S}^{\boldsymbol{x}}_i = j + 1$ and $\boldsymbol{S}^{\boldsymbol{x}}_j = i - 1$ for each substring $\boldsymbol{x}_{i\ldots j} \in L(\Sigma, \uparrow)$, where $1 \leq i < j \leq |\boldsymbol{x}|$.

Trávníček [33, Algorithm 23] proposed an algorithm constructing the subtree jump table for ordered ranked trees in the prefix *ranked* bar notation in which each node has a precomputed

a	b	a	↑	b	b	↑	↑	↑	a	b	↑	↑	↑
1	2	3	4	5	6	7	8	9	10	11	12	13	14
15	10	5	2	9	8	5	4	1	14	13	10	9	0

Figure 4.11 The subtree jump table for the prefix bar notation of the ordered labeled tree $\mathcal{T}_2$ illustrated in Figure 2.2.

arity (number of children). In Chapter 5, we propose an alternative algorithm that works directly with the prefix (unranked) bar notation of trees.

▶ Remark 4.20 (Properties of the subtree jump table for prefix bar notation). Let Σ be an alphabet. Let $\boldsymbol{x} \in L(\Sigma, \uparrow)$. Let $\boldsymbol{S}^{\boldsymbol{x}}$ be the subtree jump table for $\boldsymbol{x}$. Then, for each $i \in \{1, \dots, |\boldsymbol{x}|\}$ such that $\boldsymbol{x}_i \in \Sigma$, it holds that

- $\boldsymbol{S}^{\boldsymbol{x}}_i - 1$ is the bar position corresponding to the label at position i and
- $(\boldsymbol{S}^{\boldsymbol{x}}_i - i)/2$ is the size of the bottom-up subtree $\boldsymbol{x}_{i\dots\boldsymbol{S}^{\boldsymbol{x}}_i-1}$.

Moreover, for each $i \in \{1, \dots, |\boldsymbol{x}|\}$ such that $\boldsymbol{x}_i = \uparrow$, it holds that

- $\boldsymbol{S}^{\boldsymbol{x}}_i + 1$ is the label position corresponding to the bar at position i and
- $(i - \boldsymbol{S}^{\boldsymbol{x}}_i)/2$ is the size of the bottom-up subtree $\boldsymbol{x}_{\boldsymbol{S}^{\boldsymbol{x}}_i+1\dots i}$.

Chapter 5

Main results in inexact tree pattern matching

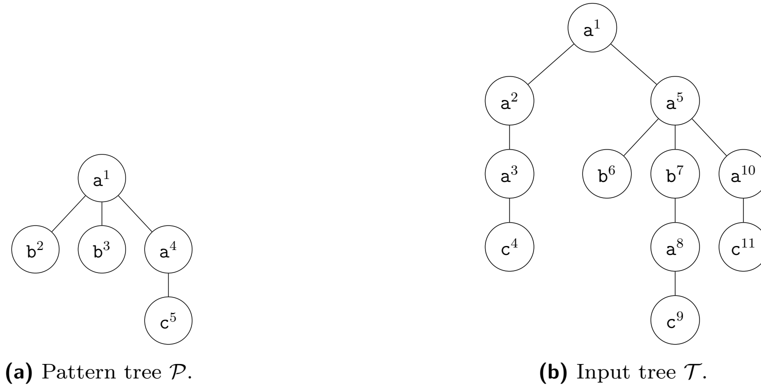
In this chapter, we focus on the problem of tree pattern matching that allows errors, also known as the inexact (or approximate) tree pattern matching. Specifically, given ordered labeled (unranked) trees $\mathcal{P}$ and $\mathcal{T}$, our goal is to find all the bottom-up subtrees of $\mathcal{T}$ that match $\mathcal{P}$ with up to k errors. To measure the number of errors, we use the 1-degree edit distance described in Section 2.3.1.2. Thus, in the SNINWE classification terminology, this chapter focuses on the NFFOBI tree pattern matching problem under the 1-degree edit distance.

Our solution is motivated and inspired by the elegant automata-based approach to the problem of inexact pattern matching in strings discussed in Section 4.1.1. In this chapter, we adapt these principles to the domain of trees and propose novel automata-based methods that solve the NFFOBI tree pattern matching problem under the 1-degree edit distance. Moreover, we show that a finite automaton is a sufficient computational model for this problem.

Before we introduce our automata-based approach to the NFFOBI tree pattern matching problem under the 1-degree edit distance, we first focus on a simpler variant of the problem under the *constrained 1-degree edit distance* in which the insertion and deletion operations cannot be used recursively to insert or delete a subtree of any size. We introduce the constrained 1-degree edit distance and its relation to the 1-degree edit distance in Section 5.1. In Section 5.2, we formally define four variants of the NFFOBI tree pattern matching problem that we aim to solve in this chapter; aside from distinguishing between the 1-degree edit distance and its constrained variant, we also distinguish between the unit cost (simple) and the non-unit cost variant. In Section 5.3, we present a general overview of our automata approach. Then, in the following three sections, we provide details of the proposed methods for the four variants of the NFFOBI tree pattern matching problem: our solution to the problem under the constrained simple 1-degree edit distance is introduced in Section 5.4, the solution to the problem under the simple 1-degree edit distance is described in Section 5.5, and finally, we propose an adaptation of our methods for the non-unit cost variant of both distances in Section 5.6. This chapter is concluded with a summary in Section 5.7.

We note that we originally presented most of the results described in this chapter as two conference papers [156], [157] and submitted them as a journal article [158]. However, this chapter explains the results in more detail and describes the relationship with the SNINWE classification. We also present slightly different variants of some algorithms.

Throughout this chapter, we demonstrate our approach on the pattern tree and the input tree illustrated in Figure 5.1. When we assume that 1-degree edit operations have non-unit costs, we use either the cost function described in Table 2.1 or the cost function described in Table 5.1. Both functions are metrics on $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \cup \{\lambda\}$.



(a) Pattern tree $\mathcal{P}$. **(b)** Input tree $\mathcal{T}$.

Figure 5.1 Two ordered labeled trees over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ that we use throughout this chapter as an example of a pattern tree and an input tree.

	a	b	c	λ
a	0	2	1	1
b	2	0	1	2
c	1	1	0	1
λ	1	2	1	0

Table 5.1 A metric cost function γ on $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \cup \{\lambda\}$ that we use throughout this chapter as an example cost function for assigning costs to 1-degree edit operations.

5.1 Constrained 1-degree edit distance

In Section 2.3.1.2, we described the 1-degree edit distance between two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ as the cost of a least-cost sequence of elementary edit operations, called 1-degree edit operations, that transform $\mathcal{T}_1$ into $\mathcal{T}_2$. In this section, we introduce a novel variant of the 1-degree edit distance, called the *constrained 1-degree edit distance*, in which we use the same elementary operations as in the case of 1-degree edit distance; however, we constrain the use of both insertion and deletion operations. We also discuss the relationship between the constrained 1-degree edit distance and the 1-degree edit distance.

The constrained use of insertion and deletion operations in the constrained 1-degree edit distance does not permit them to be used recursively to insert or delete a subtree of any size. Specifically, when transforming a tree $\mathcal{T}_1$ into a tree $\mathcal{T}_2$, only leaves initially present in $\mathcal{T}_1$ can be deleted, and only a bottom-up subtree of size one can be inserted into $\mathcal{T}_1$. For example, consider the tree $\mathcal{P}$ illustrated in Figure 5.1a. Nodes $\mathtt{b}^2, \mathtt{b}^3$, and $\mathtt{c}^5$ can be deleted during a transformation of $\mathcal{P}$ into another tree. However, we cannot delete node $\mathtt{a}^4$ since it is not initially a leaf present in $\mathcal{P}$; in other words, even if node $\mathtt{c}^5$ is deleted and node $\mathtt{a}^4$ becomes a leaf, we are still not allowed to delete it. Regarding the insertion operation, there are currently, for example, four positions where a new leaf can be inserted as a child of the root. However, if we add a new leaf v as a child of the root, we cannot add a new leaf as a child of v. In other words, new leaves can only be added as children of the nodes initially present in the tree. We refer to an edit script in which insertion and deletion operations are constrained in this way as the *constrained edit script*.

▶ **Definition 5.1** (Constrained edit script). Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be ordered labeled trees. An edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ is *constrained* if the following two conditions are satisfied:

- every deletion operation in the script refers to a node $v \in \mathsf{leaves}(\mathcal{T}_1)$ and
- every insertion operation in the script refers to a node $v \in V(\mathcal{T}_1)$.

Constrained edit scripts are particular cases of edit scripts defined in Section 2.3.1.2. However, in contrast to an edit script that always exists for any pair of ordered labeled trees, there can sometimes be no constrained edit scripts. For example, there is no constrained edit script between tree $\mathcal{P}$ and tree $\mathcal{T}$ illustrated in Figure 5.1a and Figure 5.1b, respectively.

Using constrained edit scripts, we define the *constrained 1-degree edit distance* and the *constrained simple 1-degree edit distance*.

▶ **Definition 5.2** (Constrained 1-degree edit distance). Let Σ_λ be an alphabet with the blank symbol λ equipped with a cost function. Given two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ over Σ, the *constrained 1-degree edit distance* is a function $d^{\mathrm{c}} : Tr(\Sigma) \times Tr(\Sigma) \to \mathbb{R} \cup \{\infty\}$ such that $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2)$ is either the cost of an optimal constrained edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ or ∞ if no constrained edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ exists.

▶ **Definition 5.3** (Constrained simple 1-degree edit distance). Let Σ be an alphabet. Given two ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ over Σ, the *constrained simple 1-degree edit distance* is a function $d^{\mathrm{c}}_{\mathrm{s}} : Tr(\Sigma) \times Tr(\Sigma) \to \mathbb{N}_0 \cup \{\infty\}$ such that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$ is either the length of a shortest constrained edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ or ∞ if no constrained edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ exists.

▶ **Example 5.4** (Constrained (simple) 1-degree edit distance). Let $\mathcal{P}$ be the tree illustrated in Figure 5.1a. A constrained edit script between $\mathcal{P}$ and another tree can contain only following edit operations:

- deletion of leaf $\mathtt{b}^2, \mathtt{b}^3$, and $\mathtt{c}^5$;
- insertion of new leaves as children of node $\mathtt{a}^1, \mathtt{b}^2, \mathtt{b}^3, \mathtt{a}^4$, and $\mathtt{c}^5$; and
- relabeling node $\mathtt{a}^1, \mathtt{b}^2, \mathtt{b}^3, \mathtt{a}^4$, and $\mathtt{c}^5$.

Given the bottom-up subtree $\mathcal{T}/\mathtt{a}^5$ of tree $\mathcal{T}$ illustrated in Figure 5.1b, tree $\mathcal{P}$ cannot be turned into $\mathcal{T}/\mathtt{a}^5$ using only the transformations described above. Therefore, $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{P}, \mathcal{T}/\mathtt{a}^5) = \infty$. We can, however, transform $\mathcal{P}$ into $\mathcal{T}/\mathtt{a}^5$ by inserting a new leaf labeled by $\mathtt{a}$ as a child of $\mathtt{b}^3$ followed by inserting a new leaf with label $\mathtt{c}$ as a child of the node $\mathtt{a}$ that was inserted in the previous step. Thus, $d_{\mathrm{s}}(\mathcal{P}, \mathcal{T}/\mathtt{a}^5) = 2$.

Given the bottom-up subtree $\mathcal{T}/\mathtt{b}^7$ of tree $\mathcal{T}$ illustrated in Figure 5.1b, we get $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{P}, \mathcal{T}/\mathtt{b}^7) = 3 = d_{\mathrm{s}}(\mathcal{P}, \mathcal{T}/\mathtt{b}^7)$ since $\mathcal{P}$ can be transformed into $\mathcal{T}/\mathtt{b}^7$ by deleting leaves $\mathtt{b}^2$ and $\mathtt{b}^3$ and relabeling its root to $\mathtt{b}$. Assuming that the 1-degree edit operations are assigned their costs using the cost function γ illustrated in Table 2.1, it follows that $d^{\mathrm{c}}(\mathcal{P}, \mathcal{T}/\mathtt{b}^7) = 17 = d(\mathcal{P}, \mathcal{T}/\mathtt{b}^7)$ since an optimal (constrained) edit script between $\mathcal{P}$ and $\mathcal{T}/\mathtt{b}^7$ consists of two deletion operation each of which costs 8 (leaves $\mathtt{b}^2$ and $\mathtt{b}^3$) and one relabel operation for the root ($\mathtt{a}$ to $\mathtt{b}$) that costs 1.

In the rest of this section, we discuss the relationship between the 1-degree edit distance d, the simple 1-degree edit distance d_{s}, and the constrained variants $d^{\mathrm{c}}_{\mathrm{s}}$ and d^{c}. We use these results in the following section, where we formally define the four problems we aim to solve in this chapter.

▶ **Lemma 5.5.** *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two ordered labeled trees. Assume that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) \neq \infty$. Then, $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) \geq d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$.*

Proof. To obtain a contradiction suppose that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) < d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$ for some ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$. Then the length of a shortest constrained edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ is smaller than the length of a shortest edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$. However, since every constrained edit script is also an edit script, we can use a shortest constrained edit script as a shortest edit script which contradicts that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) < d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$. ◀

In the following lemmas, we assume that the 1-degree edit operations come with a cost described by a metric cost function γ. Moreover, we assume that $\gamma(a, b) \geq 1$ for all distinct symbols $a, b \in \Sigma_\lambda$.

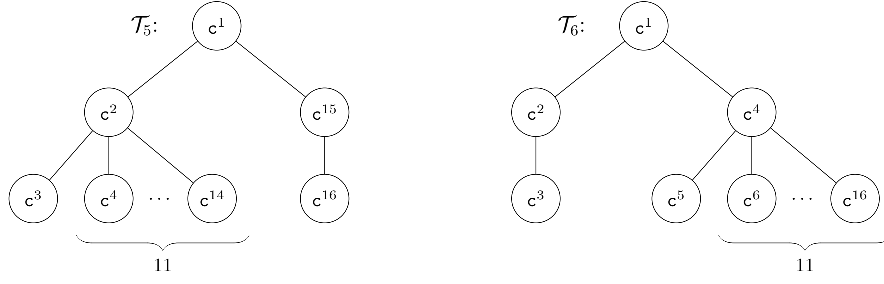


Figure 5.2 Two ordered labeled trees that we use in the proof of Lemma 5.9.

▶ **Lemma 5.6.** *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two ordered labeled trees. Assume that $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2) \neq \infty$. Then, $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2) \geq d(\mathcal{T}_1, \mathcal{T}_2)$.*

Proof. Analogue to the proof of Lemma 5.5 with a slight change that we consider the cost of an optimal (constrained) edit script instead of the length of a shortest (constrained) edit script. ◀

▶ **Lemma 5.7.** *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two ordered labeled trees. Assume that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) \neq \infty$. Then, $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2) \geq d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$.*

Proof. To obtain a contradiction suppose that $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2) < d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$ for some ordered labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$. Then the cost of an optimal constrained edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ is smaller than the length of a shortest constrained edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$. Consider an optimal constrained edit script of minimal length; we know that the costs of each operation in the script have to be greater than or equal to 1. Therefore, the length of this script has to be smaller than the length of a shortest constrained edit script, which means that we can use it as a shortest constrained edit script which contradicts $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2) < d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$. ◀

▶ **Lemma 5.8.** *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two ordered labeled trees. Then, $d(\mathcal{T}_1, \mathcal{T}_2) \geq d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$.*

Proof. Analogue to the proof of Lemma 5.7 with a slight change that we consider edit scripts instead of constrained edit scripts. ◀

▶ **Lemma 5.9.** *There exist ordered labeled trees $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4, \mathcal{T}_5$, and $\mathcal{T}_6$ and costs assigned to edit operations such that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) < d(\mathcal{T}_1, \mathcal{T}_2)$, $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_3, \mathcal{T}_4) = d(\mathcal{T}_3, \mathcal{T}_4)$, and $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_5, \mathcal{T}_6) > d(\mathcal{T}_5, \mathcal{T}_6)$.*

Proof. Assume that edit operations are assigned costs according to the cost function γ described by Table 2.1. Example 5.4 proves there exist trees $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) < d(\mathcal{T}_1, \mathcal{T}_2)$. To show that there exist trees $\mathcal{T}_3$ and $\mathcal{T}_4$ such that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_3, \mathcal{T}_4) = d(\mathcal{T}_3, \mathcal{T}_4)$, consider that $\mathcal{T}_3$ and $\mathcal{T}_4$ differ only in labels of their roots: the root of $\mathcal{T}_3$ is labeled by `a` and the root of $\mathcal{T}_4$ is labeled by `b`. An edit script that contains one relabel operation from `a` to `b` is both shortest and optimal and both its length and cost equals to 1. Finally, Figure 5.2 proves that there exist trees $\mathcal{T}_5$ and $\mathcal{T}_6$ such that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_5, \mathcal{T}_6) > d(\mathcal{T}_5, \mathcal{T}_6)$. We get $d(\mathcal{T}_5, \mathcal{T}_6) = 20$ by deleting nodes $\mathtt{c}^{16}$ and $\mathtt{c}^{15}$ and inserting two nodes labeled by `c`, and $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_5, \mathcal{T}_6) = 22$ since we need to delete 11 leaves from the first bottom-up subtree of the root and insert 11 leaves to the second. ◀

▶ **Lemma 5.10.** *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two ordered labeled trees. Assume that $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2) \neq \infty$. Then, $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2) \geq d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$.*

Proof. By Lemma 5.7, we get $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2) \geq d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$. By Lemma 5.5, we get $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2) \geq d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$. Therefore, it follows that $d^{\mathrm{c}}(\mathcal{T}_1, \mathcal{T}_2) \geq d_{\mathrm{s}}(\mathcal{T}_1, \mathcal{T}_2)$. ◀

5.2 Problem statement

In Section 3.2, we formally defined the problem of NFFOBI tree pattern matching for ordered labeled trees, in which the goal is to find all the bottom-up subtrees in an input tree that match a given pattern tree with up to k errors; see Problem 3.16. All inexact tree pattern matching problems considered in this chapter are variants of this problem, each using a different tree edit distance to measure the number of errors. As the tree edit distance we use either the constrained simple 1-degree edit distance, the simple 1-degree edit distance, the constrained 1-degree edit distance, or the 1-degree edit distance. In this section, we formally define the four variants of the NFFOBI tree pattern matching problem; and discuss their mutual relationships.

▶ Convention 5.11. In this chapter, we use k as a nonnegative integer that denotes the maximum number of errors (the maximum tree edit distance) allowed.

▶ **Problem 5.12** (NFFOBI tree pattern matching problem under d_s^c)**.** The *NFFOBI tree pattern matching problem under* d_s^c is the NFFOBI tree pattern matching problem where the constrained simple 1-degree edit distance d_s^c is used as the tree edit distance.

▶ **Problem 5.13** (NFFOBI tree pattern matching problem under d_s)**.** The *NFFOBI tree pattern matching problem under* d_s is the NFFOBI tree pattern matching problem where the simple 1-degree edit distance d_s is used as the tree edit distance.

▶ **Problem 5.14** (NFFOBI tree pattern matching problem under d^c)**.** The *NFFOBI tree pattern matching problem under* d^c is the NFFOBI tree pattern matching problem where the constrained 1-degree edit distance d^c is used as the tree edit distance.

▶ **Problem 5.15** (NFFOBI tree pattern matching problem under d)**.** The *NFFOBI tree pattern matching problem under* d is the NFFOBI tree pattern matching problem where the 1-degree edit distance d is used as the tree edit distance.

▶ **Example 5.16** (NFFOBI tree pattern matching problem under d_s^c, d_s, d^c, or d)**.** Let $\mathcal{P}$ be the pattern tree illustrated in Figure 5.1a, $\mathcal{T}$ be the input tree illustrated in Figure 5.1b, and let $k = 2$. Given $\mathcal{P}, \mathcal{T}$, and k, the solution to the NFFOBI tree pattern matching problem is as follows: First, if the constrained simple 1-degree edit distance d_s^c is used, then the solution is $\{\mathtt{a}^2\}$. That is, with respect to the maximum number of errors allowed, $\mathcal{T}/\mathtt{a}^2$ is the only occurrence of $\mathcal{P}$ in $\mathcal{T}$. Second, if the simple 1-degree edit distance d_s is used, then the solution is $\{\mathtt{a}^2, \mathtt{a}^5\}$. Finally, assume that 1-degree edit operations are assigned costs according to the cost function γ described in Table 5.1. If the constrained 1-degree edit distance d^c is used, we get the empty set of nodes as the solution. That is, there are no nodes v in the input tree such that $d^c(\mathcal{P}, \mathcal{T}/v) \leq k$. However, if we use the 1-degree edit distance d, then the solution is $\{\mathtt{a}^5\}$.

Example 5.16 suggests the existence of relationships between the problems. For example, we can see that the solution for the problem under the constrained simple 1-degree edit distance is a subset of the solution for the problem under the simple 1-degree edit distance. The following theorem describes the mutual relationships between the four variants of the NFFOBI tree pattern matching problem that we focus on in this chapter.

▶ **Theorem 5.17** (Relationships between the four variants of the NFFOBI tree pattern matching problem)**.** *Let $\mathcal{T}$ and $\mathcal{P}$ be ordered labeled trees. Let k be the maximum number of errors allowed. Let*

- $X_s^c = \{v : v \in V(\mathcal{T}) \wedge d_s^c(\mathcal{P}, \mathcal{T}/v) \leq k\}$,
- $X_s = \{v : v \in V(\mathcal{T}) \wedge d_s(\mathcal{P}, \mathcal{T}/v) \leq k\}$,
- $X^c = \{v : v \in V(\mathcal{T}) \wedge d^c(\mathcal{P}, \mathcal{T}/v) \leq k\}$, *and*

- $X = \{v : v \in V(\mathcal{T}) \wedge d(\mathcal{P}, \mathcal{T}/v) \leq k\}$.

Then, $X^{\mathrm{c}} \subseteq X^{\mathrm{c}}_{\mathrm{s}} \subseteq X_{\mathrm{s}}$ *and* $X^{\mathrm{c}} \subseteq X \subseteq X_{\mathrm{s}}$. *However,* $X^{\mathrm{c}}_{\mathrm{s}} \not\subseteq X$ *and* $X \not\subseteq X^{\mathrm{c}}_{\mathrm{s}}$ *in general.*

Proof. If $v \in X^{\mathrm{c}}$, then $d^{\mathrm{c}}(\mathcal{P}, \mathcal{T}/v) \leq k$. Applying Lemma 5.7 we can conclude that $d^{\mathrm{c}}_{\mathrm{s}}(\mathcal{P}, \mathcal{T}/v) \leq k$. That is, if $v \in X^{\mathrm{c}}$, then $v \in X^{\mathrm{c}}_{\mathrm{s}}$—in other words, $X^{\mathrm{c}} \subseteq X^{\mathrm{c}}_{\mathrm{s}}$. Using Lemma 5.5, the similar reasoning applies to the case $X^{\mathrm{c}}_{\mathrm{s}} \subseteq X_{\mathrm{s}}$. Similarly, we can prove $X^{\mathrm{c}} \subseteq X$ and $X \subseteq X_{\mathrm{s}}$ by applying Lemma 5.6 and Lemma 5.8, respectively. Finally, application of Lemma 5.9 enables us to prove that $X^{\mathrm{c}}_{\mathrm{s}} \not\subseteq X$ and $X \not\subseteq X^{\mathrm{c}}_{\mathrm{s}}$ in general. ◀

5.3 Automata approach

In this section, we give a general overview of our automata-based approach to the four variants of the NFFOBI tree pattern matching problem defined in the previous section. We base our approach on reducing the NFFOBI tree pattern matching problem to a problem of string matching. For string representation of trees, we use the prefix bar notation. Using the substring property of this notation, see Lemma 2.42, we know that if a pattern tree $\mathcal{P}$ matches a bottom-up subtree $\mathcal{S}$ of an input tree $\mathcal{T}$, then $\mathsf{prefBar}(\mathcal{P})$ matches $\mathsf{prefBar}(\mathcal{S})$, which is a substring of $\mathsf{prefBar}(\mathcal{T})$. In order to recognize that $\mathsf{prefBar}(\mathcal{P})$ matches $\mathsf{prefBar}(\mathcal{S})$, we need to be able to compute the distance between those strings that corresponds to the distance between $\mathcal{P}$ and $\mathcal{S}$. In Definition 5.18 and Theorem 5.19, we examine how 1-degree edit operations work on trees in the prefix bar notation.

▶ **Definition 5.18** (1-degree string edit operations)**.** Let Σ be an alphabet. Let $\boldsymbol{x} \in L(\Sigma, \uparrow)$. A *1-degree string edit operation* on $\boldsymbol{x}$ is either

- the *deletion* of substring $\boldsymbol{p}_i\boldsymbol{p}_{i+1}$ such that $\boldsymbol{p}_i \in \Sigma$ and $\boldsymbol{p}_{i+1} = \uparrow$, where $i \in \{2, \ldots, |\boldsymbol{p}| - 2\}$, denoted by $\mathsf{strDel}(i)$;
- the *insertion* of substring $a \uparrow$, where $a \in \Sigma$ at position $i \in \{2, \ldots, |\boldsymbol{p}|\}$, denoted by $\mathsf{strIns}(i, a)$; or
- the *substitution* (or *relabeling*) of symbol $\boldsymbol{p}_i$, where $\boldsymbol{p}_i \in \Sigma$ and $i \in \{1, \ldots, |\boldsymbol{p}| - 1\}$, for $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$, denoted by $\mathsf{strRel}(i, a)$.

▶ **Theorem 5.19** (Application of 1-degree edit operations on trees in the prefix bar notation)**.** *Let* Σ *be an alphabet. Let* $\mathcal{P} = (V, E, r, \preceq_{\mathrm{S}}, \mathsf{label})$ *be an ordered labeled tree over* Σ. *Let* $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. *The 1-degree edit operations on* $\mathcal{P}$ *correspond to 1-degree string edit operations on* $\boldsymbol{p}$ *as follows:*

- *Operation* $\mathsf{del}(v)$ *corresponds to operation* $\mathsf{strDel}(i)$*, and vice versa, where* i *is the label position of* v*.*
- *Operation* $\mathsf{ins}(a, j, u)$ *corresponds to operation* $\mathsf{strIns}(i, a)$*, and vice versa, where*

$$i = \begin{cases} 1 + \text{the label position of } u & \text{if } j = 1, \\ 1 + \text{the bar position of } (j-1)\text{-th child of } u & \text{otherwise.} \end{cases}$$

- *Operation* $\mathsf{rel}(v, a)$ *corresponds to operation* $\mathsf{strRel}(i, a)$*, and vice versa, where* i *denotes the label position of* v*.*

Proof. The proof is divided into three parts; one part for each operation.

Deletion. First, we show that $\mathsf{del}(v)$ corresponds to $\mathsf{strDel}(i)$. It follows from the definition of the prefix bar notation and the assumption that i is the label position of v that v is in $\boldsymbol{p}$ represented by substring $\mathsf{label}(v) \uparrow = \boldsymbol{p}_i\boldsymbol{p}_{i+1}$. Thus, deleting v corresponds to deleting $\boldsymbol{p}_i\boldsymbol{p}_{i+1}$. Operation $\mathsf{strDel}(i)$ deletes substring $\boldsymbol{p}_i\boldsymbol{p}_{i+1}$ if the following conditions are satisfied:

1. symbol $\boldsymbol{p}_i \in \Sigma$, symbol $\boldsymbol{p}_{i+1} = \uparrow$, and
2. $i \in \{2, \ldots, |\boldsymbol{p}| - 2\}$.

It is easy to check that the first condition is satisfied. The second condition is satisfied due to the following:

- Since node v is not the root, it follows that the label position of v is not equal to one. Thus, $i \geq 2$.
- Since node v is not the root, it also follows that $\mathcal{P}$ has at least two nodes. It is easy to see that the prefix bar notation of each tree with at least two nodes ends with at least two bar symbols. Therefore, the label position of v cannot be equal to $|\boldsymbol{p}|$ nor $|\boldsymbol{p}| - 1$. Thus, $i \leq |\boldsymbol{p}| - 2$.

Since all conditions for operation $\mathsf{strDel}(i)$ are satisfied, the operation deletes substring $\boldsymbol{p}_i\boldsymbol{p}_{i+1}$. Therefore, $\mathsf{del}(v)$ corresponds to $\mathsf{strDel}(i)$.

Now, we show that $\mathsf{strDel}(i)$ corresponds to $\mathsf{del}(v)$. Operation $\mathsf{strDel}(i)$ deletes string $\boldsymbol{p}_i\boldsymbol{p}_{i+1}$ such that $\boldsymbol{p}_i \in \Sigma$ and $\boldsymbol{p}_{i+1} = \uparrow$, where $i \in \{2, \ldots, |\boldsymbol{p}| - 2\}$. Since $\boldsymbol{p}_i \in \Sigma$ and $\boldsymbol{p}_{i+1} = \uparrow$, we get that $\boldsymbol{p}_i\boldsymbol{p}_{i+1} \in L(\Sigma, \uparrow)$. Thus, by applying Lemma 2.43, we have that $\boldsymbol{p}_i\boldsymbol{p}_{i+1}$ represents a bottom-up subtree of $\mathcal{P}$. The size of this subtree is clearly equal to one. Thus, substring $\boldsymbol{p}_i\boldsymbol{p}_{i+1}$ represents a leaf. Moreover, since $i \neq 1$ and $i \neq |\boldsymbol{p}| - 1$, it follows that the leaf is not the root. Therefore, operation $\mathsf{strDel}(i)$ corresponds to operation $\mathsf{del}(v)$, where v is the non-root leaf whose label position is i.

Insertion. First, we show that $\mathsf{ins}(a, j, u)$ corresponds to $\mathsf{strIns}(i, a)$. Operation $\mathsf{ins}(a, j, u)$ inserts a new leaf labeled by $a \in \Sigma$ as the j-th child of a node u, where $j \in \{1, \ldots, \mathsf{degree}(u) + 1\}$. The prefix bar notation of a leaf labeled by a is $a \uparrow$. Thus, inserting a leaf labeled by a into $\mathcal{P}$ corresponds to inserting substring $a \uparrow$ into $\boldsymbol{p}$. From the definition of the prefix bar notation, it follows that the label position of the first child of u is equal to 1 + the label position of u. Thus, insertion of a new leaf labeled by a as the first child of u corresponds to inserting substring $a \uparrow$ at position i that is equal to 1 + the label position of u. It is easy to see that $i \geq 2$ in this case. Moreover, the label position of the j-th child of u, where $j \geq 2$ is equal to 1 + the bar position of $(j-1)$-th child of u. Thus, insertion of a new leaf a as the j-th child of u, where $j \geq 2$ corresponds to insertion of substring $a \uparrow$ at position i that is equal to 1 + the bar position of $(j-1)$-th child of u. In this case, it also holds that $i \geq 2$. Since all conditions for operation $\mathsf{strIns}(i, a)$ are satisfied, the operation inserts substring $a \uparrow$ at position i that corresponds to the label position of the j-th child of u. Therefore, $\mathsf{ins}(a, j, u)$ corresponds to $\mathsf{strIns}(i, a)$.

Now, we show that $\mathsf{strIns}(i, a)$ corresponds to $\mathsf{ins}(a, j, u)$. Operation $\mathsf{strIns}(i, a)$ inserts substring $a \uparrow$, where $a \in \Sigma$ at position $i \in \{2, \ldots, |\boldsymbol{p}|\}$. Since $a \uparrow$ belongs to $L(\Sigma, \uparrow)$, we have that this substring represents a tree. Clearly, size of this tree is equal to one. Thus, insertion of substring $a \uparrow$ corresponds to insertion of a leaf into the underlying tree. If $\boldsymbol{p}_{i-1} \in \Sigma$, then from the definition of the prefix bar notation follows that $i - 1$ is the label position of a node that is the parent of the new inserted leaf. Thus, the new leaf is the first child of its parent. This operation on $\boldsymbol{p}$ corresponds to operation $\mathsf{ins}(a, 1, u)$, where u is the node whose label position is $i - 1$. If $\boldsymbol{p}_{i-1} = \uparrow$, then from the definition of the prefix bar notation follows that $i - 1$ is the bar position of the sibling that immediately precedes the new inserted leaf. Thus, the new leaf is j-th child of its parent, where $j \geq 2$. Therefore, this operation on $\boldsymbol{p}$ corresponds to operation $\mathsf{ins}(a, j, u)$, where the bar position of $(j-1)$-th child of u is $i - 1$.

Relabeling. First, we show that $\mathsf{rel}(v, a)$ corresponds to $\mathsf{strRel}(i, a)$. From the assumption that i is the label position of v, we have that $\boldsymbol{p}_i = \mathsf{label}(v)$. Thus, relabeling node v corresponds to substitution of symbol $\boldsymbol{p}_i$. Operation $\mathsf{strRel}(i, a)$ can be applied if the following conditions are satisfied:

1. $\boldsymbol{p}_i \in \Sigma$,

2. $i \in \{1, \dots, |\boldsymbol{p}| - 1\}$, and
3. $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$.

The first condition and third conditions are clearly satisfied. The second condition is also satisfied because the label position cannot be equal to $|\boldsymbol{p}|$ (there is always the bar symbol at that position). Since all conditions for operation $\mathsf{strRel}(i, a)$ are satisfied, the operation substitutes symbol $\boldsymbol{p}_i$ for a. Therefore, $\mathsf{rel}(v, a)$ corresponds to $\mathsf{strRel}(i, a)$.

Now, we show that $\mathsf{strRel}(i, a)$ corresponds to $\mathsf{rel}(v, a)$. Operation $\mathsf{strRel}(i, a)$ substitutes symbol $\boldsymbol{p}_i$, where $\boldsymbol{p}_i \in \Sigma$ and $i \in \{1, \dots, |\boldsymbol{p}| - 1\}$, for $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$. Since $\boldsymbol{p}_i \in \Sigma$, it follows from the definition and properties of the prefix bar notation that symbol $\boldsymbol{p}_i$ corresponds to the label of a node whose label position is i. Therefore, operation $\mathsf{strRel}(i, a)$ corresponds to operation $\mathsf{rel}(v, a)$, where v is the node whose label position is i. ◀

▶ **Example 5.20** (Application of 1-degree edit operations on trees in the prefix bar notation)**.** Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the ordered labeled trees illustrated in Figure 2.3. The figure also shows that we can transform $\mathcal{T}_1$ into $\mathcal{T}_2$ using a sequence of five 1-degree edit operations. Assume that both $\mathcal{T}_1$ and $\mathcal{T}_2$ are represented as strings in the prefix bar notation, that is

- $\mathsf{prefBar}(\mathcal{T}_1) = \mathsf{abc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow \mathsf{aa} \uparrow\uparrow\uparrow$, and
- $\mathsf{prefBar}(\mathcal{T}_2) = \mathsf{caa} \uparrow\uparrow \mathsf{bc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow\uparrow$.

Then, the edit script between $\mathcal{T}_1$ and $\mathcal{T}_2$ illustrated in Figure 2.3 corresponds to the following sequence of 1-degree string edit operations:

$$\begin{aligned}
\mathsf{abc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow \mathsf{aa} \uparrow\uparrow\uparrow &\xrightarrow{\text{strRel(1,c)}} \mathsf{cbc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow \mathsf{aa} \uparrow\uparrow\uparrow \\
\mathsf{cbc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow \mathsf{aa} \uparrow\uparrow\uparrow &\xrightarrow{\text{strDel(13)}} \mathsf{cbc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow \mathsf{a} \uparrow\uparrow \\
\mathsf{cbc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow \mathsf{a} \uparrow\uparrow &\xrightarrow{\text{strDel(12)}} \mathsf{cbc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow\uparrow \\
\mathsf{cbc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow\uparrow &\xrightarrow{\text{strIns(2,a)}} \mathsf{ca} \uparrow \mathsf{bc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow\uparrow \\
\mathsf{ca} \uparrow \mathsf{bc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow\uparrow &\xrightarrow{\text{strIns(3,a)}} \mathsf{caa} \uparrow\uparrow \mathsf{bc} \uparrow \mathsf{c} \uparrow \mathsf{c} \uparrow \mathsf{a} \uparrow\uparrow\uparrow
\end{aligned}$$

Let $\boldsymbol{t_1}, \boldsymbol{t_2} \in L(\Sigma, \uparrow)$. We define an *edit script* between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ similarly as we did for trees in Section 2.3.1.2, that is, as a sequence of 1-degree string edit operations that transform $\boldsymbol{t}_1$ into $\boldsymbol{t}_2$. Analogously, the *length* of an edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ is the number of operations in the script, and a *shortest edit script* between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ is an edit script with the minimum number of operations required to transform $\boldsymbol{t}_1$ into $\boldsymbol{t}_2$. We use the length of a shortest edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ to define the *simple 1-degree string edit distance.*

▶ **Definition 5.21** (Simple 1-degree string edit distance)**.** Let Σ be an alphabet. Given two strings $\boldsymbol{t}_1, \boldsymbol{t}_2 \in L(\Sigma, \uparrow)$, the *simple 1-degree string edit distance* is a function $d_{\mathrm{s}} : L(\Sigma, \uparrow) \times L(\Sigma, \uparrow) \to \mathbb{N}_0$ such that $d_{\mathrm{s}}(\boldsymbol{t}_1, \boldsymbol{t}_2)$ is the length of a shortest edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$.

A cost function γ defined on Σ_λ can be used to define the *cost of 1-degree string edit operations* as follows: Let $(a, b) \in \Sigma_\lambda \times \Sigma_\lambda$. If $a \neq b \neq \lambda$, then $\gamma(a, b)$ corresponds to the cost of substituting symbol a for symbol b. If $a = \lambda$ and $b \neq \lambda$, then $\gamma(a, b)$ corresponds to the cost of inserting substring $b \uparrow$. Finally, if $a \neq \lambda$ and $b = \lambda$, then $\gamma(a, b)$ denotes the cost of deleting substring $a \uparrow$. Similarly as for the cost of 1-degree (tree) edit operations, the pairs $(a, a) \in \Sigma_\lambda \times \Sigma_\lambda$ do not correspond to any 1-degree string edit operation and thus do not play a role in defining their costs. *Cost of an edit script* between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ is the sum of costs of operations in the script, and an *optimal edit script* between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ is an edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ with the minimal cost. We use the cost of an optimal edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ to define the *1-degree string edit distance.*

▶ **Definition 5.22** (1-degree string edit distance)**.** Let Σ_λ be an alphabet with the blank symbol λ equipped with a cost function. Given two strings $\boldsymbol{t}_1, \boldsymbol{t}_2 \in L(\Sigma, \uparrow)$, the *1-degree string edit distance* is a function $d_\text{s} : L(\Sigma, \uparrow) \times L(\Sigma, \uparrow) \to \mathbb{R}$ such that $d_\text{s}(\boldsymbol{t}_1, \boldsymbol{t}_2)$ is the cost of an optimal edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$.

From Theorem 5.19, it follows that we can convert every edit script between strings $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ from $L(\Sigma, \uparrow)$ to an edit script between the underlying trees, and vice versa. From this, we conclude that $d_\text{s}(\boldsymbol{t_1}, \boldsymbol{t_2}) = d_\text{s}(\mathcal{T}_1, \mathcal{T}_2)$ and $d(\boldsymbol{t_1}, \boldsymbol{t_2}) = d(\mathcal{T}_1, \mathcal{T}_2)$ for every pair $\boldsymbol{t}_1, \boldsymbol{t}_2 \in L(\Sigma, \uparrow)$ and trees $\mathcal{T}_1, \mathcal{T}_2 \in \mathit{Tr}(\Sigma)$ such that $\boldsymbol{t}_1 = \mathsf{prefBar}(\mathcal{T}_1)$ and $\boldsymbol{t}_2 = \mathsf{prefBar}(\mathcal{T}_2)$. Moreover, we say that an edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ is *constrained* if the edit script between the underlying trees is constrained. We use constrained edit scripts to define constrained versions for the (simple) 1-degree string edit distance.

▶ **Definition 5.23** (Constrained 1-degree string edit distance)**.** Let Σ_λ be an alphabet with the blank symbol λ equipped with a cost function. Given two strings $\boldsymbol{t}_1, \boldsymbol{t}_2 \in L(\Sigma, \uparrow)$, the *constrained 1-degree string edit distance* is a function $d_\text{s} : L(\Sigma, \uparrow) \times L(\Sigma, \uparrow) \to \mathbb{R} \cup \{\infty\}$ such that $d_\text{s}(\boldsymbol{t}_1, \boldsymbol{t}_2)$ is either the cost of an optimal constrained edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ or ∞ if no constrained edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ exist.

▶ **Definition 5.24** (Constrained simple 1-degree string edit distance)**.** Let Σ be an alphabet. Given two strings $\boldsymbol{t}_1, \boldsymbol{t}_2 \in L(\Sigma, \uparrow)$, the *constrained simple 1-degree string edit distance* is a function $d_\text{s} : L(\Sigma, \uparrow) \times L(\Sigma, \uparrow) \to \mathbb{N}_0 \cup \{\infty\}$ such that $d_\text{s}(\boldsymbol{t}_1, \boldsymbol{t}_2)$ is either the length of a shortest edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ or ∞ if no constrained edit script between $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ exists.

We can see that $d_\text{s}^\text{c}(\boldsymbol{t_1}, \boldsymbol{t_2}) = d_\text{s}^\text{c}(\mathcal{T}_1, \mathcal{T}_2)$ and $d^\text{c}(\boldsymbol{t_1}, \boldsymbol{t_2}) = d^\text{c}(\mathcal{T}_1, \mathcal{T}_2)$ for every pair $\boldsymbol{t}_1, \boldsymbol{t}_2 \in L(\Sigma, \uparrow)$ and trees $\mathcal{T}_1, \mathcal{T}_2 \in \mathit{Tr}(\Sigma)$ such that $\boldsymbol{t}_1 = \mathsf{prefBar}(\mathcal{T}_1)$ and $\boldsymbol{t}_2 = \mathsf{prefBar}(\mathcal{T}_2)$.

We can now reduce the NFFOBI tree pattern matching to a string matching problem as follows:

- Using the prefix bar notation, encode pattern tree $\mathcal{P}$ over Σ and input tree $\mathcal{T}$ over Σ as strings $\boldsymbol{p}$ and $\boldsymbol{t}$ from $L(\Sigma, \uparrow)$, respectively.
- Instead of tree edit distance function f, where f is $d_\text{s}, d_\text{s}^\text{c}, d$, or d^c, use the corresponding string edit distance function f.
- Find every substring $\boldsymbol{t}_{j' \dots j}$ of $\boldsymbol{t}$, where $1 \le j' < j \le |\boldsymbol{t}|$, such that $\boldsymbol{t}_{j' \dots j} \in L(\Sigma, \uparrow)$ and $f(\boldsymbol{p}, \boldsymbol{t}_{j' \dots j}) \le k$, where k is the given maximum number of errors allowed.

Specifically, we focus on finding end positions of occurrences of $\boldsymbol{p}$ in $\boldsymbol{t}$. Every end position corresponds to the bar position of a unique node $v \in V(\mathcal{T})$ such that $f(\mathcal{P}, \mathcal{T}/v) \le k$.

Given a pattern tree $\mathcal{P}$ and a maximum number of allowed errors k, the main idea of our automata-based approach is to identify the *pattern dictionary*, the set of all strings representing the trees whose distance from $\mathcal{P}$ is at most k. Then, we built a dictionary automaton called the *1-degree matching automaton*.

▶ **Definition 5.25** (Pattern dictionary)**.** Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Let $k \ge 0$. Let f be equal to either $d_\text{s}, d_\text{s}^\text{c}, d$, or d^c. A *pattern dictionary* for $\boldsymbol{p}, k$ and f, denoted by $L(\boldsymbol{p}, k, f)$, is a (string) language defined as follows:

$$L(\boldsymbol{p}, k, f) = \{\boldsymbol{p}' : \boldsymbol{p}' \in L(\Sigma, \uparrow) \wedge f(\boldsymbol{p}, \boldsymbol{p}') \le k\} .$$

▶ **Definition 5.26** (1-degree matching automaton)**.** Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Let $k \ge 0$. Let f be equal to $d_\text{s}, d_\text{s}^\text{c}, d$, or d^c. A *1-degree matching automaton* for $\boldsymbol{p}, k$, and f, denoted by $\mathcal{M}(\boldsymbol{p}, k, f)$, is a string automaton accepting the following language:

$$\left\{\boldsymbol{x}\boldsymbol{p}' : \boldsymbol{x} \in \Sigma_\uparrow^* \wedge \boldsymbol{p}' \in L(\boldsymbol{p}, k, f)\right\} .$$

The 1-degree matching automaton accepts an infinite language. It can read (not necessarily accept) any ordered labeled tree in the prefix bar notation. To find the positions of all the occurrences of the pattern tree in a given input tree $\mathcal{T}$, the automaton is run on $\mathsf{prefBar}(\mathcal{T})$. The automaton then reports a match every time it goes through a final state. In this way, the automaton works similarly to a transducer, where we output a flag that signals an occurrence every time we follow a transition into a final state. Algorithm 5.27 describes the process. If the automaton is deterministic and the time necessary to compute a transition from each state is constant, then all pattern occurrences are located in time linear to the size of the input.

▶ **Algorithm 5.27** (String automata approach to the NFFOBI tree pattern matching problem)**.** Let Σ be an alphabet. Let $\mathcal{P}$ and $\mathcal{T}$ be ordered labeled trees over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$ and $\boldsymbol{t} = \mathsf{prefBar}(\mathcal{T})$. Let $k \geq 0$. Let f be equal to $d_{\mathrm{s}}, d_{\mathrm{s}}^{\mathrm{c}}, d$, or d^{c}. Assuming that the 1-degree matching automaton $\mathcal{M}(\boldsymbol{p}, k, f)$ is given, the algorithm uses it to find every bottom-up subtree $\mathcal{S}$ in $\mathcal{T}$ such that $f(\mathcal{P}, \mathcal{S}) \leq k$.

1. (Look for occurrences.) Using $\mathcal{M}(\boldsymbol{p}, k, f)$, read each symbol $\boldsymbol{t}_j$ in $\boldsymbol{t}$: If a final state is reached, then output j.

In the following sections, we present the construction of the 1-degree matching automaton. Moreover, we discuss that the automaton can also be used to report the number of errors for each occurrence.

5.4 1-degree matching automaton for the constrained simple 1-degree edit distance

The NFFOBI tree pattern matching problem under the constrained simple 1-degree edit distance is the simplest of the four problems defined in Section 5.2. Let us recall that the constrained simple 1-degree edit distance between two ordered labeled trees is given by the length of a shortest constrained edit script between them (or ∞ if no such script exists). In other words, deletion and insertion operations in edit scripts can only refer to nodes initially present in the tree, and we assume that all edit operations come with the unit cost.

In this section, we present our solution to the NFFOBI tree pattern matching problem under $d_{\mathrm{s}}^{\mathrm{c}}$ that is based on a string automaton. We do this by reducing Problem 5.12 to a string matching problem, as described in the previous section. First, we show that the 1-degree automaton can be constructed as an ε-NFA. Then, we discuss its deterministic variant and show a simulation of the ε-NFA by dynamic programming.

Let $\mathcal{P}$ be a pattern tree. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Let $k \geq 0$. We describe the construction of the 1-degree matching ε-NFA for $d_{\mathrm{s}}^{\mathrm{c}}$ in two parts. First, we present an algorithm that constructs an ε-NFA accepting the pattern dictionary $L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})$; see Algorithm 5.28. Then, we transform it into the 1-degree matching ε-NFA; see Algorithm 5.30. Both automata have $|\boldsymbol{p}| + 1 + k\,(2|\boldsymbol{p}| - 1)$ states and can be built in time $\mathcal{O}(k \cdot |\boldsymbol{p}|)$.

▶ **Algorithm 5.28** (Construction of an ε-NFA accepting a pattern dictionary for $d_{\mathrm{s}}^{\mathrm{c}}$)**.** Let Σ be an alphabet. Let $k \geq 0$. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Given $\boldsymbol{p}$ and k, the algorithm constructs an ε-NFA $\mathcal{M} = (Q, \Sigma_{\uparrow}, \delta, q_0, F)$ accepting language $L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})$, the pattern dictionary for $\boldsymbol{p}$, k, and $d_{\mathrm{s}}^{\mathrm{c}}$.

1. (Define the set of states.) Each state is labeled by i^l or i_1^l, where $i \in \{0, \dots, |\boldsymbol{p}|\}$ and $l \in \{0, \dots, k\}$. Specifically, states are labeled so they can be arranged into rows and columns as illustrated in Figure 5.3. We say that a state labeled by i^l (or i_1^l) it is at *depth* i and at *level* l; states labeled by i_1^l are called *auxiliary.*

$$\text{Set } Q \leftarrow \left\{0^0\right\} \cup \left\{i^l : 1 \leq i \leq |\boldsymbol{p}| \wedge 0 \leq l \leq k\right\} \cup \underbrace{\left\{i_1^l : 1 \leq i \leq |\boldsymbol{p}| - 1 \wedge 1 \leq l \leq k\right\}}_{\text{auxiliary states}}.$$

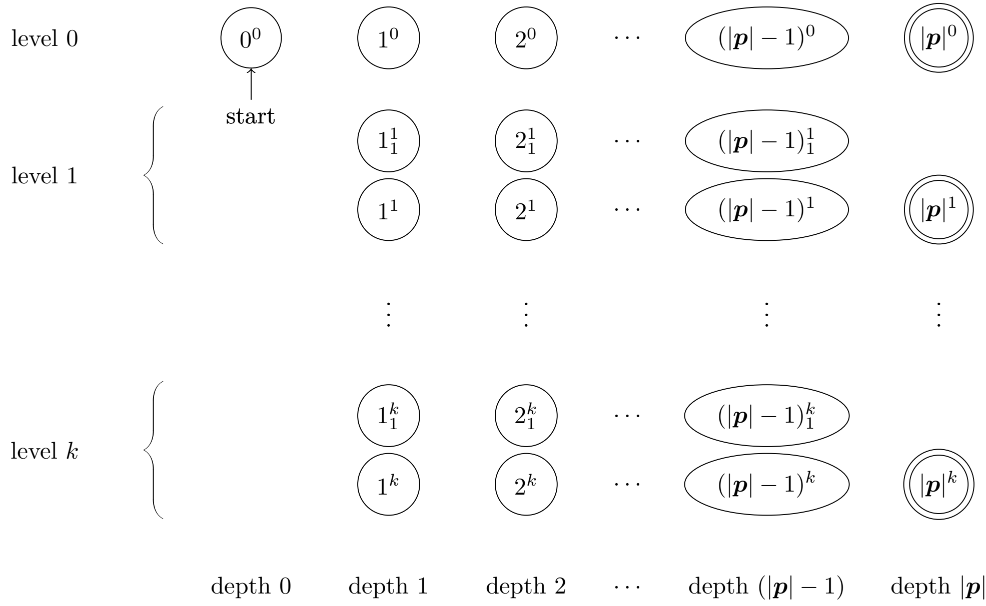


Figure 5.3 The regular structure of the ε-NFA constructed by Algorithm 5.28. Given $k \geq 1$ and a pattern string $\boldsymbol{p} \in L(\Sigma, \uparrow)$, the set of states can be arranged into rows and columns. There are $(k+1)$ *main rows*, one for each level $l \in \{0, \dots, k\}$, and k *auxiliary rows*, one for each level greater than or equal to 1. Auxiliary rows are placed between two main rows. Each main row, except the one for level 0, is composed of $|\boldsymbol{p}|$ states with depth $1, 2, \dots, |\boldsymbol{p}|$. In the row for level 0, there are $|\boldsymbol{p}|+1$ states of depth $0, 1, 2, \dots, |\boldsymbol{p}|$. Each auxiliary row is composed of $|\boldsymbol{p}|-1$ states of depth $1, 2, \dots, |\boldsymbol{p}|-1$.

2. (Define the start state.) Set $q_0 \leftarrow 0^0$.

3. (Define the set of final states.) Every state at depth $|\boldsymbol{p}|$ is final. That is, every main row ends in one of the final states.

$$\text{Set } F \leftarrow \left\{|\boldsymbol{p}|^l : 0 \leq l \leq k\right\}.$$

4. (Add transitions indicating match.)
   a. Set $\delta\left(0^0, \boldsymbol{p}_1\right) \leftarrow \left\{1^0\right\}$.
   b. For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \dots, k\}$, set $\delta\left((i-1)^l, \boldsymbol{p}_i\right) \leftarrow \left\{i^l\right\}$.

5. (Add transitions for relabeling operations.)
   a. If $k \geq 1$, then for each $a \in \Sigma \setminus \{\boldsymbol{p}_1\}$, set $\delta\left(0^0, a\right) \leftarrow \left\{1^1\right\}$.
   b. For each position $i \in \{2, \dots, |\boldsymbol{p}|-1\}$ and number of errors $l \in \{0, \dots, k-1\}$: If $\boldsymbol{p}_i \neq \uparrow$, then for each $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$, set $\delta\left((i-1)^l, a\right) \leftarrow \left\{i^{l+1}\right\}$.

6. (Add transitions for insertion operations.) Insertion operations are modelled by auxiliary states and two consecutive transitions: insertion of $a \in \Sigma$ and the corresponding bar symbol.

   For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$, number of errors $l \in \{0, \dots, k-1\}$, and symbol $a \in \Sigma$:
   a. Set $\delta\left((i-1)^l, a\right) \leftarrow \delta\left((i-1)^l, a\right) \cup \left\{(i-1)^{l+1}_1\right\}$.
   b. Set $\delta\left((i-1)^{l+1}_1, \uparrow\right) \leftarrow \left\{(i-1)^{l+1}\right\}$.

7. (Add transitions for deletion operations.) For each position $i \in \{2, \ldots, |\boldsymbol{p}| - 2\}$ and number of errors $l \in \{0, \ldots, k-1\}$: If $\boldsymbol{p}_i \in \Sigma$ and $\boldsymbol{p}_{i+1} = \uparrow$, then set $\delta\left((i-1)^l, \varepsilon\right) \leftarrow \left\{(i+1)^{l+1}\right\}$.
8. (Return.) Return $\mathcal{M} = (Q, \Sigma_\uparrow, \delta, q_0, F)$.

▶ **Lemma 5.29** (Correctness of Algorithm 5.28)**.** *Given an ordered labeled tree $\mathcal{P}$ and $k \geq 0$, the ε-NFA $\mathcal{M}$ constructed by Algorithm 5.28 accepts the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d_\mathrm{s}^\mathrm{c})$.*

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. The procedure is to show that $L(\mathcal{M}) = L(\boldsymbol{p}, k, d_\mathrm{s}^\mathrm{c})$. Thus, the proof falls naturally into two steps.

($\subseteq$) Assume $\boldsymbol{x} \in L(\mathcal{M})$. Thus, there exists $l \in \{0, \ldots, k\}$ such that $|\boldsymbol{p}|^l \in \hat{\delta}\left(0^0, \boldsymbol{x}\right)$ and there is no $l' < l$ such that $|\boldsymbol{p}|^{l'} \in \hat{\delta}\left(0^0, \boldsymbol{x}\right)$. We show that $\boldsymbol{x} \in L(\boldsymbol{p}, k, d_\mathrm{s}^\mathrm{c})$ by showing that $d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{x}) = l$. We use induction on the maximum number of errors k.

- Assume $k = 0$. Then, $|\boldsymbol{p}|^0 \in \hat{\delta}\left(0^0, \boldsymbol{x}\right)$. Thus, $\mathcal{M}$ reads $\boldsymbol{x}$ using transitions created in Step 4. Therefore, $\boldsymbol{x} = \boldsymbol{p}$, and it follows that $d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{x}) = 0$ and $\boldsymbol{x} \in L(\boldsymbol{p}, k, d_\mathrm{s}^\mathrm{c})$.
- Assume $k \geq 1$ and that the claim holds for errors smaller than k. For each $\boldsymbol{x} \in L(\mathcal{M})$, there exists $l \in \{0, \ldots, k\}$ such that $|\boldsymbol{p}|^l \in \hat{\delta}\left(0^0, \boldsymbol{x}\right)$ and there is no $l' < l$ such that $|\boldsymbol{p}|^{l'} \in \hat{\delta}\left(0^0, \boldsymbol{x}\right)$. For $l \in \{0, \ldots, k-1\}$, we get that $d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{x}) = l$ by the induction hypothesis. Our goal is to show that if $|\boldsymbol{p}|^k \in \hat{\delta}\left(0^0, \boldsymbol{x}\right)$, where for no $k' < k$ holds that $|\boldsymbol{p}|^{k'} \in \hat{\delta}\left(0^0, \boldsymbol{x}\right)$, then $d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{x}) = k$. To accept $\boldsymbol{x}$ in state $|\boldsymbol{p}|^k$, automaton $\mathcal{M}$ must use exactly k transitions created in Steps 5, 6, or 7 (for simplicity, we view the pair of transitions created in Step 6 as one transition). Let $\boldsymbol{y}$ be the shortest prefix of $\boldsymbol{x}$ such that $\mathcal{M}$ gets to a state at level $k-1$ after reading $\boldsymbol{y}$. From that state, we can continue to state $|\boldsymbol{p}|^{k-1}$ using a sequence of transitions created in Step 4. It is easy to check that this sequence corresponds to reading a nonempty suffix $\boldsymbol{z}$ of $\boldsymbol{p}$. Thus, $|\boldsymbol{p}|^{k-1} \in \hat{\delta}\left(0^0, \boldsymbol{yz}\right)$. Since there is no $k' < k$ such that $|\boldsymbol{p}|^{k'} \in \hat{\delta}\left(0^0, \boldsymbol{x}\right)$, there is also no $l < k-1$ such that $|\boldsymbol{p}|^l \in \hat{\delta}\left(0^0, \boldsymbol{yz}\right)$ (otherwise, we could set $k' = l + 1$). Therefore, we get that $d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{yz}) = k - 1$ by the induction hypothesis. Since $\boldsymbol{p}$ can written as $\boldsymbol{p} = \boldsymbol{wz}$, we get that each of the $k-1$ edit operations were applied to $\boldsymbol{w}$ in order to change it to $\boldsymbol{y}$. To read $\boldsymbol{x}$, automaton $\mathcal{M}$ reads $\boldsymbol{y}$ followed by string $\boldsymbol{z}'$. We need to show that $\boldsymbol{z}$ can be transformed to $\boldsymbol{z}'$ using one edit operation. To read $\boldsymbol{z}'$, automaton uses a sequence of transitions created in Step 4 (at level $k-1$) followed by a transition created in Steps 5, 6, or 7 and a sequence of transitions created in Step 4 (at level k). Using a transition created in Steps 5, 6, or 7 corresponds to using an operation relabeling, insertion, or deletion on $\boldsymbol{z}$, respectively. It follows that $d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{x}) = k$. Therefore, $\boldsymbol{x} \in L(\boldsymbol{p}, k, d_\mathrm{s}^\mathrm{c})$, and the claim holds.

($\supseteq$) Assume $\boldsymbol{x} \in L(\boldsymbol{p}, k, d_\mathrm{s}^\mathrm{c})$. Thus, there exists $l \in \{0, \ldots, k\}$ such that $d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{x}) = l$. We show that $\boldsymbol{x} \in L(\mathcal{M})$ by showing that $|\boldsymbol{p}|^l \in \hat{\delta}\left(0^0, \boldsymbol{x}\right)$. We use induction on the maximum number of errors k.

- Assume $k = 0$. Then, $L(\boldsymbol{p}, k, d_\mathrm{s}^\mathrm{c}) = \{\boldsymbol{p}\}$. From Step 4 of the algorithm, it follows that $|\boldsymbol{p}|^0 \in \hat{\delta}\left(0^0, \boldsymbol{p}\right)$. From Step 3, we get that $|\boldsymbol{p}|^0$ is a final state. Thus, $\boldsymbol{p} \in L(\mathcal{M})$.
- Assume $k \geq 1$ and that the claim holds for errors smaller than k. Then, $L(\boldsymbol{p}, k, d_\mathrm{s}^\mathrm{c}) = L(\boldsymbol{p}, k-1, d_\mathrm{s}^\mathrm{c}) \cup \{\boldsymbol{y} : \boldsymbol{y} \in L(\Sigma, \uparrow) \wedge d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{y}) = k\}$. By the induction hypothesis, we have that $L(\boldsymbol{p}, k-1, d_\mathrm{s}^\mathrm{c}) \subseteq L(\mathcal{M})$. Our goal is to show that $\{\boldsymbol{y} : \boldsymbol{y} \in L(\Sigma, \uparrow) \wedge d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{y}) = k\} \subseteq L(\mathcal{M})$. We know that for each such $\boldsymbol{y}$, there is a constrained edit script between $\boldsymbol{p}$ and $\boldsymbol{y}$ with k operations. These operations can be sorted from left to right according to which position in $\boldsymbol{p}$ they refer to. This way we can transform $\boldsymbol{p}$ to $\boldsymbol{y}$ by traversing $\boldsymbol{p}$ from its beginning to its end. If we remove the rightmost operation from such a script, then we clearly transform $\boldsymbol{p}$ to string $\boldsymbol{y}'$, where $d_\mathrm{s}^\mathrm{c}(\boldsymbol{p}, \boldsymbol{y}') = k - 1$. For each such $\boldsymbol{y}'$, we have that $|\boldsymbol{p}|^{k-1} \in \hat{\delta}\left(0^0, \boldsymbol{y}'\right)$ by the induction hypothesis. It is easy to check that there is a nonempty suffix of $\boldsymbol{p}$ that is intact

(not changed by any of the first $k-1$ edit operations) whose length depends on the set of the $k-1$ operations that transformed $\boldsymbol{p}$ to $\boldsymbol{y}'$. Similarly, since $|\boldsymbol{p}|^{k-1} \in \hat{\delta}(0^0, \boldsymbol{y}')$, the automaton must at some point go to a state at level $k-1$ from which only transitions created in Step 4 are used. Now we apply the operation that has been removed. This operation can be one of the following:

- The deletion of substring $\boldsymbol{p}_i\boldsymbol{p}_{i+1}$ such that $\boldsymbol{p}_i \in \Sigma$ and $\boldsymbol{p}_{i+1} = \uparrow$, where $i \in \{2, \dots, |\boldsymbol{p}|-2\}$. In the automaton, a symbol $\boldsymbol{p}_i$ is read by transitions leading from a state at depth $i-1$ to a state at depth i. Thus, deletion of substring $\boldsymbol{p}_i\boldsymbol{p}_{i+1}$ corresponds to ε-transition that leads from a state at depth $i-1$ to a state at depth $i+1$. These transitions are created in Step 7 of the algorithm. Moreover, the state level is increased by 1. After that, the automaton reads the rest of the string using transitions created in Step 4 and accepts it by the final state $|\boldsymbol{p}|^k$.
- The substitution of symbol $\boldsymbol{p}_i$, where $\boldsymbol{p}_i \in \Sigma$ and $i \in \{1, \dots, |\boldsymbol{p}|-1\}$, for $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$. In the automaton, this corresponds to transitions that increase level by 1 and lead from a state at depth $i-1$ to a state at depth i. These transitions are created in Step 5 of the algorithm. After that, the automaton reads the rest of the string using transitions created in Step 4 and accepts it by the final state $|\boldsymbol{p}|^k$.
- The insertion of substring $a\uparrow$, where $a \in \Sigma$, at position $i \in \{2, \dots, |\boldsymbol{p}|\}$. In the automaton, this corresponds to pairs of transitions that increase level by 1 and do not change the state depth. These transitions are created in Step 6 of the algorithm. After that, the automaton reads the rest of the string using transitions created in Step 4 and accepts it by the final state $|\boldsymbol{p}|^k$.

Since for every operation there is a corresponding transition (or a pair of transitions) that also increases the state level by one, we have that $|\boldsymbol{p}|^k \in \hat{\delta}(0^0, \boldsymbol{y})$ for every $\boldsymbol{y} \in L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})$, where $d_{\mathrm{s}}^{\mathrm{c}}(\boldsymbol{p}, \boldsymbol{y}) = k$. Hence, the claim holds. ◀

We are now in a position to construct the 1-degree matching ε-NFA for $d_{\mathrm{s}}^{\mathrm{c}}$. We do this by first constructing the ε-NFA accepting the pattern dictionary, and then adding a loop transition to its start state that enables the automaton to read the prefix bar notation of any input tree. See Algorithm 5.30.

We show an example of the ε-NFA constructed by Algorithm 5.28 and the 1-degree matching ε-NFA constructed by Algorithm 5.30 in Figure 5.4. Insertion operations are represented by two consecutive *vertical* transitions that go through an auxiliary state; *diagonal* ε-transitions represent deletion operations, and the remaining *diagonal* transitions represent relabeling operations.

▶ **Algorithm 5.30** (Construction of 1-degree matching ε-NFA for $d_{\mathrm{s}}^{\mathrm{c}}$)**.** Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $k \geq 0$. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Given $\boldsymbol{p}$ and k, the algorithm constructs the 1-degree matching ε-NFA for $\boldsymbol{p}, k$, and $d_{\mathrm{s}}^{\mathrm{c}}$.

1. (Construct automaton for the pattern dictionary.) Construct an ε-NFA $(Q, \Sigma_\uparrow, \delta, q_0, F)$ accepting the pattern dictionary $L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})$ using Algorithm 5.28.
2. (Add loop to the start state.) For each symbol $a \in \Sigma_\uparrow$, set $\delta(q_0, a) \leftarrow \delta(q_0, a) \cup \{q_0\}$.
3. (Return.) Return $\mathcal{M} = (Q, \Sigma_\uparrow, \delta, q_0, F)$.

▶ **Theorem 5.31** (Correctness of Algorithm 5.30)**.** *Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $k \geq 0$. Given* $\mathsf{prefBar}(\mathcal{P})$ *and k, Algorithm 5.30 constructs the 1-degree matching ε-NFA for* $\mathsf{prefBar}(\mathcal{P}), k$, *and $d_{\mathrm{s}}^{\mathrm{c}}$.*

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Let $\mathcal{M}$ be the automaton that is returned by the algorithm in Step 3. We show that $L(\mathcal{M}) = \{\boldsymbol{x}\boldsymbol{p}' : \boldsymbol{x} \in \Sigma_\uparrow^* \wedge \boldsymbol{p}' \in L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})\}$. In Step 1, we create an

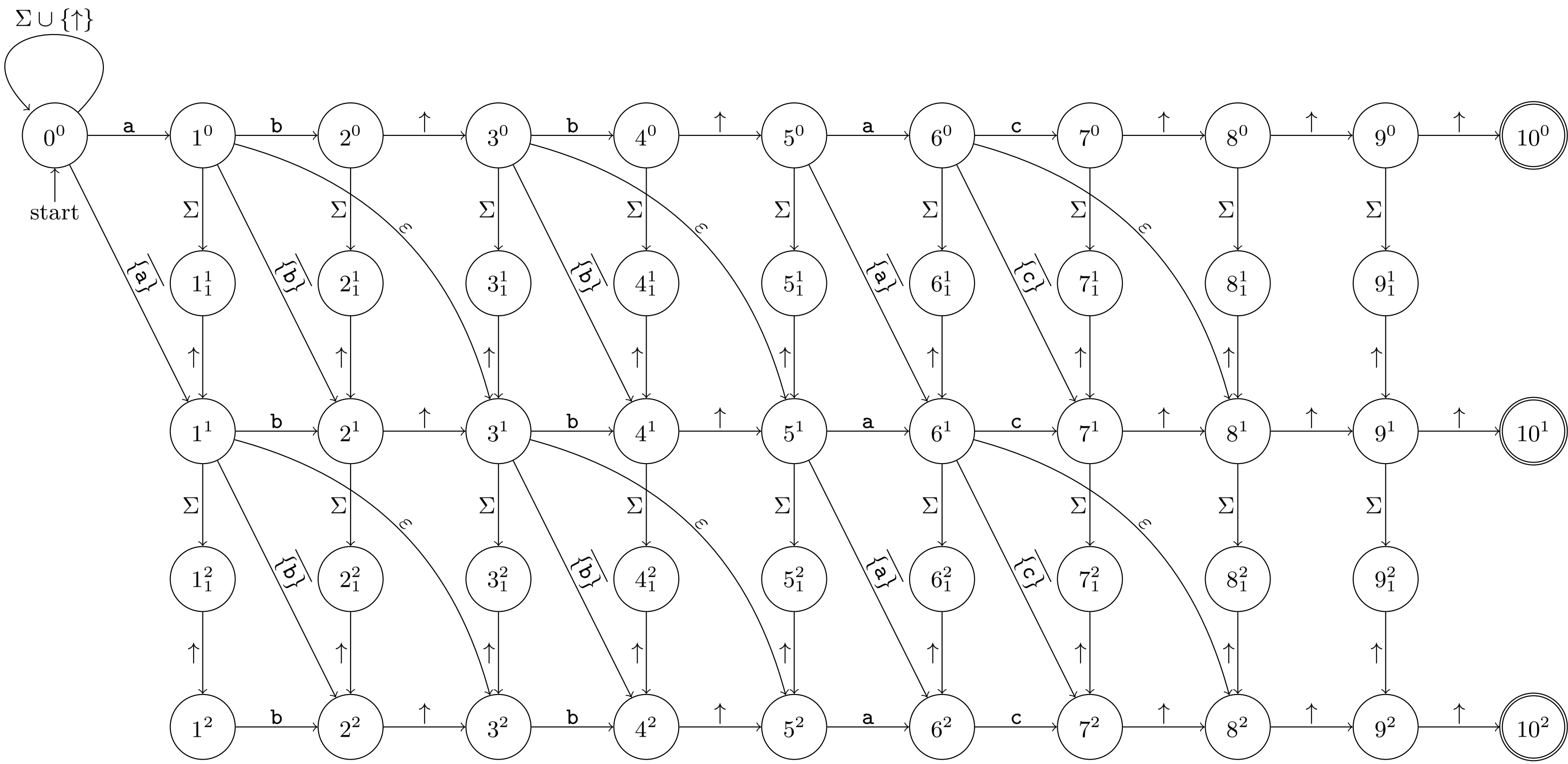


Figure 5.4 The ε-NFA constructed by Algorithm 5.28 accepting the pattern dictionary or the 1-degree matching ε-NFA constructed by Algorithm 5.30 for distance d_s^c, maximum number of errors $k = 2$, and pattern tree $\mathcal{P}$ illustrated in Figure 5.1a. If the automaton contains loop transitions in the start state, it is the 1-degree matching ε-NFA $\mathcal{M}(\mathsf{prefBar}(\mathcal{P}), k, d_s^c)$. If the loop transitions in the start state are removed, it is the ε-NFA accepting the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d_s^c)$.

ε-NFA accepting language $\{\boldsymbol{p}' : \boldsymbol{p}' \in L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})\}$. The correctness of this automaton is given by Lemma 5.29. In Step 2, we add a loop transition to its start state for each symbol $a \in \Sigma_{\uparrow}$. Thus, the automaton can read arbitrary many symbols and stay at the start state. At any point, the nondeterminism allows the automaton to proceed to another state and read string $\boldsymbol{p}'$ such that $d_{\mathrm{s}}^{\mathrm{c}}(\boldsymbol{p}, \boldsymbol{p}') \leq k$. This automaton is returned in Step 3 as automaton $\mathcal{M}$. Therefore, $L(\mathcal{M}) = \big\{\boldsymbol{x}\boldsymbol{p}' : \boldsymbol{x} \in \Sigma_{\uparrow}^{*} \wedge \boldsymbol{p}' \in L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})\big\}$. ◀

There are two ways how to use the automaton constructed by Algorithm 5.30 as a basis for a matching algorithm. First, we can transform it into an equivalent DFA to obtain a linear-time searching algorithm (assuming that the time necessary to compute a transition from each state is constant). However, this approach comes with a high state complexity; see Section 5.4.1. An alternative approach using less space but with higher time complexity for searching is based on simulating the 1-degree matching ε-NFA. In Section 5.4.2, we propose such a simulation algorithm based on dynamic programming.

5.4.1 Deterministic automaton

The 1-degree matching ε-NFA can be turned into an equivalent DFA by eliminating ε-transitions and using the subset construction. If the time necessary to compute a transition from each state is constant, then the deterministic automaton can locate all pattern occurrences in time linear to the size of the input tree.

▶ **Theorem 5.32** (Time complexity of the searching phase for the 1-degree matching DFA for $d_{\mathrm{s}}^{\mathrm{c}}$). *Let $\mathcal{T}$ and $\mathcal{P}$ be ordered labeled trees. Assuming that the time necessary to compute a transition from each state is $\mathcal{O}(1)$, the 1-degree matching DFA for $d_{\mathrm{s}}^{\mathrm{c}}$ solves Problem 5.12 in time $\Theta(|\mathcal{T}|)$.*

Proof. String $\mathsf{prefBar}(\mathcal{T})$ is in the searching phase read exactly once, symbol by symbol from left to right. The appropriate transition is taken each time a symbol is read, resulting in exactly $2|\mathcal{T}|$ transitions. An occurrence of $\mathcal{P}$ is reported every time the DFA enters a final state. ◀

Given a pattern tree over alphabet Σ with m nodes and $k \geq 0$, the state complexity of the 1-degree matching DFA is trivially $\mathcal{O}\big(2^{mk}\big)$ since the 1-degree matching (ε-)NFA has $\Theta(mk)$ states. In the rest of this section, we prove a non-trivial upper bound on the state complexity of the 1-degree matching DFA that is $\mathcal{O}\big(|\Sigma|^{k} \cdot k \cdot m^{k+1}\big)$. We do so by using the previous results by Crochemore and Melichar concerning the dictionary automaton that we discussed in Section 4.1.1.2. Recall that Theorem 4.6 shows that the state complexity of a deterministic dictionary automaton that is created from an (ε-)NFA accepting dictionary D by adding the loop transitions to its start state and using the subset construction, is $\mathcal{O}\big(\sum_{\boldsymbol{w} \in D} |\boldsymbol{w}|\big)$. Therefore, the 1-degree matching DFA for $d_{\mathrm{s}}^{\mathrm{c}}$ has $\mathcal{O}\big(\sum_{\boldsymbol{p}' \in L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})} |\boldsymbol{p}'|\big)$ states, where $L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})$ is the pattern dictionary for pattern tree $\mathcal{P}$, where $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$, maximum number of errors k, and distance $d_{\mathrm{s}}^{\mathrm{c}}$. Hence, our goal is to determine the length of the pattern dictionary. We do so by first discussing its size; that is, the number of ordered labeled trees whose distance from $\mathcal{P}$ is at most k.

▶ **Lemma 5.33** (Number of ordered labeled trees created by relabeling operations). *Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ with m nodes. The number of strings where each represents the prefix bar notation of an ordered labeled tree over Σ that is created from* $\mathsf{prefBar}(\mathcal{P})$ *by at most k relabeling operations is $\mathcal{O}\big(|\Sigma|^{k} m^{k}\big)$.*

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. The set of strings created from $\boldsymbol{p}$ by l relabeling operations, where $0 \leq l \leq k \leq m$, is made by substituting l symbols $a \in \Sigma$ in $\boldsymbol{p}$ for other symbols. There are $\binom{m}{l}$ possibilities for choosing l such symbols from $\boldsymbol{p}$ and $|\Sigma| - 1$ possibilities for choosing the new symbol. Hence, the number of such strings is at most $\binom{m}{l}(|\Sigma| - 1)^{l} = \mathcal{O}\big(m^{l}\big)(|\Sigma| - 1)^{l} = \mathcal{O}\big(|\Sigma|^{l} m^{l}\big)$. Therefore, the number of strings where each represents the prefix bar notation of an ordered labeled tree over Σ that is created from $\boldsymbol{p}$ by at most k relabeling operations is $\sum_{l=0}^{k} \mathcal{O}\big(|\Sigma|^{l} m^{l}\big) = \mathcal{O}\big(|\Sigma|^{k} m^{k}\big)$. ◀

In the following two lemmas, we use the notion of a *constrained* deletion operation and a *constrained* insertion operation. We call a deletion operation constrained if it can be included in a constrained edit script. In other words, the constrained deletion operation refers only to leaves initially present in a given tree. Similarly, we call an insertion operation constrained if it can be included in a constrained edit script; that is, the insertion operation refers only to nodes initially present in a given tree.

▶ **Lemma 5.34** (Number of ordered labeled trees created by constrained deletion operations). *Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ with m nodes. The number of strings where each represents the prefix bar notation of an ordered labeled tree over Σ that is created from* $\mathsf{prefBar}(\mathcal{P})$ *by at most k constrained deletion operations is* $\mathcal{O}\big(|\,\mathsf{leaves}(\mathcal{P})|^k\big)$.

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. The set of strings created from $\boldsymbol{p}$ by l constrained deletion operations, where $0 \leq l \leq k \leq |\,\mathsf{leaves}(\mathcal{P})|$, is made by deleting l substrings $a\uparrow$, where $a \in \Sigma$ from $\boldsymbol{p}$. There are $\binom{|\,\mathsf{leaves}(\mathcal{P})|}{l} = \mathcal{O}\big(|\,\mathsf{leaves}(\mathcal{P})|^l\big)$ possibilities for choosing l such substrings from $\boldsymbol{p}$. Therefore, the number of strings where each represents the prefix bar notation of an ordered labeled tree over Σ that is created from $\boldsymbol{p}$ by at most k constrained deletion operations is $\sum_{l=0}^{k} \mathcal{O}\big(|\,\mathsf{leaves}(\mathcal{P})|^l\big) = \mathcal{O}\big(|\,\mathsf{leaves}(\mathcal{P})|^k\big)$. ◀

▶ **Lemma 5.35** (Number of ordered labeled trees created by constrained insertion operations). *Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ with m nodes. The number of strings where each represents the prefix bar notation of an ordered labeled tree over Σ that is created from* $\mathsf{prefBar}(\mathcal{P})$ *by at most k constrained insertion operations is* $\mathcal{O}\big(|\Sigma|^k m^k\big)$.

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. The set of strings created from $\boldsymbol{p}$ by l constrained insertion operations, where $0 \leq l \leq k$ corresponds to the set of *start-final dipaths* that can be found in the ε-NFA constructed by Algorithm 5.28 in which all the relabeling and constrained deletion transitions are removed. A start-final dipath is a sequence of transitions that starts in the start state, ends in a final state, and respects the transition orientations. There are $2m$ steps needed to be done to the right (reading the pattern) and l steps down (constrained insertion operations; the two transitions representing a constrained insertion operation can be viewed as one step). Hence, every start-final dipath can be represented by a string containing $2m$ letters R (right) and l letters D (down). Moreover, every start-final dipath must start and end with letter R. It follows that the number of start-final dipaths is $\binom{2m-2+l}{l}$. Since each letter D can represent $|\Sigma|$ inserted substrings $a\uparrow$, where $a \in \Sigma$, we conclude that the number of strings created from $\boldsymbol{p}$ by exactly l constrained insertion operations is at most $\binom{2m+l-2}{l}|\Sigma|^l = \mathcal{O}\big(m^l|\Sigma|^l\big)$. Therefore, the number of strings where each represents the prefix bar notation of an ordered labeled tree over Σ that is created from $\boldsymbol{p}$ by at most k constrained insertion operations is $\sum_{l=0}^{k} \mathcal{O}\big(|\Sigma|^l m^l\big) = \mathcal{O}\big(|\Sigma|^k m^k\big)$. ◀

▶ **Lemma 5.36** (Size of the pattern dictionary). *Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ with m nodes. The size the pattern dictionary* $L(\mathsf{prefBar}(\mathcal{P}), k, d_{\mathrm{s}}^{\mathrm{c}})$ *is* $\mathcal{O}\big(|\Sigma|^k m^k\big)$.

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. The size of the pattern dictionary $L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})$ corresponds to the number of strings created from $\boldsymbol{p}$ by at most k edit operations where each edit operation is either relabeling, constrained deletion, or constrained insertion. Using Lemmas 5.33, 5.34, and 5.35, it follows that this number can be computed for $i + j + l \leq k$ as follows:

$$\begin{aligned}\overbrace{\mathcal{O}\big(|\Sigma|^i m^i\big)}^{\text{relabel } i \text{ nodes}}\ \overbrace{\mathcal{O}\big(|\Sigma|^j m^j\big)}^{\text{insert } j \text{ leaves}}\ \overbrace{\mathcal{O}\big(|\,\mathsf{leaves}(\mathcal{P})|^l\big)}^{\text{delete } l \text{ leaves}} &= \mathcal{O}\big(|\Sigma|^{i+j} m^{i+j}|\,\mathsf{leaves}(\mathcal{P})|^l\big)\\ \mathcal{O}\big(|\Sigma|^{i+j} m^{i+j}|\,\mathsf{leaves}(\mathcal{P})|^l\big) &= \mathcal{O}\big(|\Sigma|^{i+j} m^{i+j+l}\big) = \mathcal{O}\big(|\Sigma^k|m^k\big).\end{aligned}$$

Therefore, $|L(\boldsymbol{p}, k, d_{\mathrm{s}}^{\mathrm{c}})| = \mathcal{O}\big(|\Sigma^k|m^k\big)$. ◀

▶ **Theorem 5.37** (Number of states of the 1-degree matching DFA for $d_{\mathrm{s}}^{\mathrm{c}}$). *Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ with m nodes. Let $k \geq 0$ be the maximum number of*

errors allowed. The number of states of the 1-degree matching DFA $\mathcal{M}(\mathsf{prefBar}(\mathcal{P}), k, d_s^c)$ that is obtained (by eliminating ε-transitions and using the subset construction) from the 1-degree matching ε-NFA constructed by Algorithm 5.30 is $\mathcal{O}(|\Sigma|^k \cdot k \cdot m^{k+1})$.

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. The 1-degree matching DFA is constructed using three steps: (1) Using Algorithm 5.28, construct the ε-NFA that accepts the pattern dictionary $L(\boldsymbol{p}, k, d_s^c)$. (2) Add a loop transition to the start state for every symbol $a \in \Sigma \cup \{\uparrow\}$. (3) Remove ε-transitions and use the subset construction to obtain the 1-degree matching DFA. It follows from Theorem 4.6 that the number of states of such an automaton is at most the length of the pattern dictionary. Since for every string $\boldsymbol{p}' \in L(\boldsymbol{p}, k, d_s^c)$, we have that $|\boldsymbol{p}'| \leq 2m + 2k$, the length of language $L(\boldsymbol{p}, k, d_s^c)$ is

$$\mathcal{O}\Big(\sum_{\boldsymbol{p}' \in L(\boldsymbol{p},k,d_s^c)} |\boldsymbol{p}'|\Big) = \mathcal{O}\big((2m + 2k) \cdot |\Sigma|^k \cdot m^k\big) = \mathcal{O}\big(|\Sigma|^k \cdot k \cdot m^{k+1}\big).$$

Therefore, the number of states of the 1-degree matching DFA $\mathcal{M}(\boldsymbol{p}, k, d_s^c)$ that is obtained (by eliminating ε-transitions and using the subset construction) from the 1-degree matching ε-NFA constructed by Algorithm 5.30 is $\mathcal{O}(|\Sigma|^k \cdot k \cdot m^{k+1})$. ◀

5.4.2 Simulation by dynamic programming

As we mentioned in Section 4.1.2.1, dynamic programming is the most often used approach for solving the tree edit distance problem as well as the inexact tree pattern matching problem. In this section, we present a novel dynamic programming approach to the NFFOBI tree pattern matching problem under d_s^c. Moreover, we connect this approach to our automata-based solution by showing that it can be seen as a simulation of the 1-degree matching ε-NFA. This approach comes with a better space complexity than the 1-degree matching DFA but at the cost of worse time complexity for searching.

Our dynamic programming approach to the NFFOBI tree pattern matching problem under d_s^c is similar to the dynamic programming approach for strings described in Section 4.1.1.1. However, in contrast to string matching where at each mismatched position any edit operation substitution, deletion, or insertion is possible, there exist positions for trees in the prefix bar notation where only some edit operations are possible. For example, we cannot use relabeling operation when either the current symbol in the pattern or the input is equal to the bar symbol. We present our dynamic programming approach in Algorithm 5.38.

▶ **Algorithm 5.38** (Dynamic programming approach to the NFFOBI tree pattern matching problem under d_s^c.)**.** Let Σ be an alphabet. Let $\mathcal{P}$ and $\mathcal{T}$ be ordered labeled trees over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$ and $\boldsymbol{t} = \mathsf{prefBar}(\mathcal{T})$. Let $k \geq 0$. Given $\boldsymbol{p}, \boldsymbol{t}$, and k, the algorithm finds every substring $\boldsymbol{t}_{j'\dots j}$ of $\boldsymbol{t}$, where $1 \leq j' < j \leq |\boldsymbol{t}|$, such that $\boldsymbol{t}_{j'\dots j} \in L(\Sigma, \uparrow)$ and $d_s^c(\boldsymbol{p}, \boldsymbol{t}_{j'\dots j}) \leq k$. In other words, the algorithm finds every bottom-up subtree $\mathcal{S}$ in $\mathcal{T}$ such that $d_s^c(\mathcal{P}, \mathcal{S}) \leq k$.

The principal internal data structure is a two-dimensional array $\boldsymbol{D}$ in which the dimensions have ranges 0 to $|\boldsymbol{p}|$ and 0 to $|\boldsymbol{t}|$, respectively. By $\boldsymbol{D}_{i,j}$, where $i \in \{0, \dots |\boldsymbol{p}|\}$ and $j \in \{0, \dots, |\boldsymbol{t}|\}$, we denote the value at *row* i and *column* j. The value $\boldsymbol{D}_{i,j}$ with $i, j \geq 1$, depends on three positions at most: $\boldsymbol{D}_{i-1,j-1}$, $\boldsymbol{D}_{i-2,j}$, and $\boldsymbol{D}_{i,j-2}$ (see Figure 5.5). When the array is filled, $\boldsymbol{D}_{|\boldsymbol{p}|,j}$ contains $d_s^c(\boldsymbol{p}, \boldsymbol{t}_{j'\dots j})$, where $j' \in \{1, \dots, j-1\}$, if and only if $\boldsymbol{t}_{j'\dots j}$ represents the prefix bar notation of a bottom-up subtree of $\mathcal{T}$.

1. (Initialize array.)
   a. Set $\boldsymbol{D}_{0,0} \leftarrow 0$.
   b. For each position i in $\boldsymbol{p}$, set $\boldsymbol{D}_{i,0} \leftarrow \infty$.
   c. For each position j in $\boldsymbol{t}$, set $\boldsymbol{D}_{0,j} \leftarrow 0$.

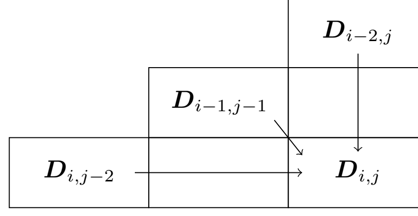


Figure 5.5 The value $\boldsymbol{D}_{i,j}$ with $i, j \geq 1$, depends on three positions at most: $\boldsymbol{D}_{i-1,j-1}$, $\boldsymbol{D}_{i-2,j}$, and $\boldsymbol{D}_{i,j-2}$.

2. (Fill array.) For each position j in $\boldsymbol{t}$ and i in $\boldsymbol{p}$:

   a. The cost of relabeling $\boldsymbol{p}_i$ to $\boldsymbol{t}_j$ (or match) is

$$c_1 \leftarrow \begin{cases} \boldsymbol{D}_{i-1,j-1} & \text{if } \boldsymbol{p}_i = \boldsymbol{t}_j, \\ \boldsymbol{D}_{i-1,j-1} + 1 & \text{if } \boldsymbol{p}_i \neq \boldsymbol{t}_j \wedge \boldsymbol{p}_i, \boldsymbol{t}_j \neq \uparrow, \\ \infty & \text{otherwise.} \end{cases}$$

   b. The cost of deleting substring $\boldsymbol{p}_{i-1}\boldsymbol{p}_i$ from $\boldsymbol{p}$ is

$$c_2 \leftarrow \begin{cases} \boldsymbol{D}_{i-2,j} + 1 & \text{if } i \geq 3 \wedge \boldsymbol{p}_i = \uparrow \wedge \boldsymbol{p}_{i-1} \neq \uparrow, \\ \infty & \text{otherwise.} \end{cases}$$

   c. The cost of inserting substring $\boldsymbol{t}_{j-1}\boldsymbol{t}_j$ into $\boldsymbol{p}$ (at position $i+1$) is

$$c_3 \leftarrow \begin{cases} \boldsymbol{D}_{i,j-2} + 1 & \text{if } j \geq 3 \wedge \boldsymbol{t}_j = \uparrow \wedge \boldsymbol{t}_{j-1} \neq \uparrow \wedge i < |\boldsymbol{p}|, \\ \infty & \text{otherwise.} \end{cases}$$

   d. Set $\boldsymbol{D}_{i,j} \leftarrow \mathsf{min}(c_1, c_2, c_3)$. (We assume that $\infty > c$ for every $c \in \mathbb{N}_0$, also that $\infty + 1 = \infty$, and that $\mathsf{min}(\infty, \infty, \infty) = \infty$.)

3. (Report occurrences.) For each position j in $\boldsymbol{t}$: If $\boldsymbol{D}_{|\boldsymbol{p}|,j} \leq k$, output j.

We show an example of our dynamic programming approach to the NFFOBI tree pattern matching problem in Figure 5.6. Note that if the last row contains a number, then the corresponding position in the input string always contains the bar symbol. This is because the position corresponds to the end position of a pattern occurrence, and each such occurrence is a bottom-up subtree of the input tree whose encoding always ends with the bar symbol. For similar reasons, each occurrence always starts at a position that does not contain the bar symbol. The start position for each occurrence can be easily computed in time linear to its size: we start at the end position of the occurrence and walk backward in the input string, increasing a counter each time we see the bar symbol and decreasing it otherwise; as soon as the counter reaches 0, the current position is the start position of the occurrence. The start positions can also be precomputed beforehand during the transformation of the input tree to its prefix bar notation. Moreover, if the original input tree structure is available, we can connect each bar symbol in its prefix bar notation to the corresponding node. Thus, instead of reporting end positions in the input string as occurrences, we can return the root nodes of the corresponding bottom-up subtrees.

In Figure 5.6, we also illustrate how edit scripts can be computed. We can obtain a corresponding edit script between pattern tree $\mathcal{P}$ and each occurrence of $\mathcal{P}$ in a given input tree $\mathcal{T}$ by tracking back the array from position $\boldsymbol{D}_{|\boldsymbol{p}|,j} \neq \infty$, where $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$, to position $\boldsymbol{D}_{0,j'-1}$ such that $\boldsymbol{t}_{j'\ldots j}$ represents the prefix bar notation of a bottom-up subtree of $\mathcal{T}$, where $\boldsymbol{t} = \mathsf{prefBar}(\mathcal{T})$.

$\boldsymbol{D}$	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
i	-	-	a	a	a	c	↑	↑	↑	a	b	↑	b	a	c	↑	↑	↑	a	c	↑	↑	↑	↑
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	a	∞	0	0	0	1	1	∞	∞	0	1	1	1	0	1	1	∞	∞	0	1	1	∞	∞	∞
2	b	∞	∞	1	1	1	2	∞	∞	∞	0	∞	1	2	1	3	∞	∞	∞	1	∞	∞	∞	∞
3	↑	∞	1	1	1	2	1	2	∞	1	2	0	2	1	2	1	3	∞	1	2	1	∞	∞	∞
4	b	∞	∞	2	2	2	3	∞	∞	∞	1	∞	0	3	2	4	∞	∞	∞	2	∞	∞	∞	∞
5	↑	∞	2	2	2	3	2	3	∞	2	3	1	3	2	3	2	4	∞	2	3	2	∞	∞	∞
6	a	∞	∞	2	2	3	3	∞	∞	∞	3	∞	2	3	3	4	∞	∞	∞	3	∞	∞	∞	∞
7	c	∞	∞	∞	3	2	4	∞	∞	∞	∞	∞	∞	3	3	4	∞	∞	∞	∞	∞	∞	∞	∞
8	↑	∞	∞	3	3	4	2	4	∞	∞	4	∞	3	4	4	3	4	∞	∞	4	∞	∞	∞	∞
9	↑	∞	∞	∞	∞	∞	4	2	4	∞	∞	4	∞	∞	∞	4	3	4	∞	∞	4	∞	∞	∞
10	↑	∞	∞	∞	∞	∞	∞	4	2	∞	∞	∞	∞	∞	∞	∞	4	3	∞	∞	∞	4	∞	∞

Figure 5.6 An example of dynamic programming approach to the NFFOBI tree pattern matching problem under $d_{\mathrm{s}}^{\mathrm{c}}$ that finds all occurrences of pattern tree $\mathcal{P}$ illustrated in Figure 5.1a in input tree $\mathcal{T}$ illustrated in Figure 5.1b with up to k errors. For $k = 2$, there is one occurrence of $\mathcal{P}$ in $\mathcal{T}$; the one found at position 7 of $\boldsymbol{t} = \mathsf{prefBar}(\mathcal{T})$. The corresponding edit script can be obtained by tracking back the array from $\boldsymbol{D}_{10,7}$ to $\boldsymbol{D}_{0,j'-1}$ such that $\boldsymbol{t}_{j'\ldots 7}$ represents the prefix bar notation of a bottom-up subtree of $\mathcal{T}$. In this case, the tracking goes back to $\boldsymbol{D}_{0,1}$. The substring $\boldsymbol{t}_{2\ldots 7}$ corresponds to the bottom-up subtree $\mathcal{T}/\mathtt{a}^2$, and the edit script between $\mathcal{P}$ and $\mathcal{T}/\mathtt{a}^2$ consists of two deletion operations.

Note that multiple paths can exist. For each occurrence $\boldsymbol{t}_{j'\ldots j}$, where $1 \leq j' < j \leq |\boldsymbol{t}|$, this computation takes $\mathcal{O}(|\boldsymbol{p}| + |\boldsymbol{t}_{j'\ldots j}|)$ additional time and space.

We prove the correctness of our matching algorithm by showing that it is a simulation of the 1-degree matching ε-NFA for $d_{\mathrm{s}}^{\mathrm{c}}$.

▶ **Theorem 5.39** (Correctness of Algorithm 5.38)**.** *The dynamic programming approach described by Algorithm 5.38 is a simulation of the 1-degree matching ε-NFA for $d_{\mathrm{s}}^{\mathrm{c}}$ constructed by Algorithm 5.30.*

Proof. Let Σ be an alphabet. Let $\mathcal{P}$ and $\mathcal{T}$ be ordered labeled trees over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$ and $\boldsymbol{t} = \mathsf{prefBar}(\mathcal{T})$. Let $k \geq 0$. We recall that in the 1-degree matching ε-NFA for $\boldsymbol{p}$, k, and $d_{\mathrm{s}}^{\mathrm{c}}$, each state is labeled by i^l or i_1^l, where $i \in \{0, \ldots, |\boldsymbol{p}|\}$ and $l \in \{0, \ldots, k\}$. Specifically, states are labeled so they can be arranged into rows and columns as illustrated in Figure 5.3. We say that a state labeled by i^l (or i_1^l) it is at depth i and at level l; states labeled by i_1^l are called auxiliary. We also recall that the dynamic programming algorithm constructs a two-dimensional array $\boldsymbol{D}$ in which the dimensions have ranges 0 to $|\boldsymbol{p}|$ and 0 to $|\boldsymbol{t}|$, respectively. By $\boldsymbol{D}_{i,j}$, where $i \in \{0, \ldots |\boldsymbol{p}|\}$ and $j \in \{0, \ldots, |\boldsymbol{t}|\}$, we denote the value at row i and column j.

We prove that every value $\boldsymbol{D}_{i,j} \leq k$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j of the run of the ε-NFA. For $\boldsymbol{D}_{i,j}$ that contains ∞ or a number greater than k, we show that no non-auxiliary state is active in depth i of the ε-NFA and step j of the run of the ε-NFA. From this proof, it follows that $\boldsymbol{D}_{|\boldsymbol{p}|,j} \leq k$ if and only if the final state $|\boldsymbol{p}|^{\boldsymbol{D}_{|\boldsymbol{p}|,j}}$ is active in step j of the run of the ε-NFA and there is no $l < \boldsymbol{D}_{|\boldsymbol{p}|,j}$ such that state $|\boldsymbol{p}|^l$ is also active. In other words, the dynamic programming reports an occurrence with distance $l \in \{0, \ldots, k\}$ ending at position j in $\boldsymbol{t}$ if and only if the ε-NFA reports an occurrence with distance l ending at position j in $\boldsymbol{t}$.

We prove the claim mentioned above by a loop invariant proof using the following invariant: Before the start of iteration j' of the outer loop, for each $i \in \{0, \ldots, |\boldsymbol{p}|\}$ and $j'' \in \{0, \ldots, j' - 1\}$,

value $\boldsymbol{D}_{i,j''}$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA is active.

Before the iteration of the first pass of the outer loop, $\boldsymbol{D}_{i,0}$ is initialized to 0 for $i = 0$ (Step 1a) and to ∞ for each $i \in \{1, \dots, |\boldsymbol{p}|\}$ (Step 1b). At step 0 of the run of the ε-NFA, only state 0^0 is active. Thus, the claim holds for $j' = 1$.

If the loop invariant is true before the start of iteration j' of the outer loop, we show that it is true before the iteration $j' + 1$. Before the iteration j', we get that for each $i \in \{0, \dots, |\boldsymbol{p}|\}$ and $j'' \in \{0, \dots, j' - 1\}$, value $\boldsymbol{D}_{i,j''}$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA is active.

During iteration j' the inner loop is executed. We prove that for the inner loop, the following invariant holds: Before the start of iteration i' of the inner loop, for each $i'' \in \{0, \dots, i' - 1\}$, value $\boldsymbol{D}_{i'',j'}$ corresponds to the level of the topmost active non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA is active.

Before the iteration of the first pass of the inner loop, $\boldsymbol{D}_{0,j'}$ should contain the level of the topmost active non-auxiliary state in depth 0 of the ε-NFA and step j' of the run of the ε-NFA. At each step $j' \in \{0, \dots, |\boldsymbol{t}|\}$ of the run of the ε-NFA, state 0^0 is active due to its loop transitions. Thus, the level of the topmost active non-auxiliary state in depth 0 of the ε-NFA and step j' of the run of the ε-NFA is 0. This corresponds to setting value $\boldsymbol{D}_{0,j'}$ to 0 in Step 1c for each $j' \in \{1, \dots, |\boldsymbol{t}|\}$ and setting value $\boldsymbol{D}_{0,0}$ to 0 in Step 1a.

If the invariant is true before the start of iteration i' of the inner loop, we show that it is true before the iteration $i' + 1$. Before the start of iteration i' of the inner loop, we get that for each $i'' \in \{0, \dots, i' - 1\}$, value $\boldsymbol{D}_{i'',j'}$ corresponds to the level of the topmost active non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA is active.

During iteration i', Step 2d is executed, and $\boldsymbol{D}_{i',j'}$ is set to $\mathsf{min}(c_1, c_2, c_3)$. We need to show that if there exists an active non-auxiliary state in depth i' of the ε-NFA and step j' of the run of the ε-NFA, then the level of the topmost one is equal to c_1, c_2, or c_3; otherwise, if there is no active non-auxiliary state in depth i' of the ε-NFA and step j' of the run of the ε-NFA, then c_1, c_2, and c_3 are equal to ∞ or contain value greater than k.

There are at most four types of transitions that lead to each non-auxiliary state in depth $i' \in \{1, \dots, |\boldsymbol{p}|\}$ of the ε-NFA:

- a match transition (created in Step 4 of Algorithm 5.28),
- a relabeling transition (created in Step 5 of Algorithm 5.28),
- an insertion transition (created in Step 6 of Algorithm 5.28; we can see the two consecutive transitions leading through an auxiliary state as one transition), and
- a deletion transition (created in Step 7 of Algorithm 5.28).

Therefore, if there is an active non-auxiliary state in depth i' of the ε-NFA and step j' of the run of the ε-NFA, we get to it using the above mentioned transitions. However, not all transitions are always possible:

- The match transitions leading to depth i' can be used in step j' of the run of the ε-NFA only if there is an active non-auxiliary state in depth $i' - 1$ and step $j' - 1$ of the run of the ε-NFA and $\boldsymbol{t}_{j'} = \boldsymbol{p}_{i'}$.
- The relabeling transitions leading to depth i' can be used in step j' of the run of the ε-NFA only if there is an active non-auxiliary state in depth $i' - 1$ and step $j' - 1$ of the run of the ε-NFA and $\boldsymbol{t}_{j'} \neq \boldsymbol{p}_{i'}$ and $\boldsymbol{p}_{i'}, \boldsymbol{t}_{j'} \neq \uparrow$.

- The deletion transitions leading to depth i' for $i' \geq 3$ can be used in step j' of the run of the ε-NFA only if there is an active non-auxiliary state in depth $i'-2$ and step j' of the run of the ε-NFA and $\boldsymbol{p}_{i'} = \uparrow$ and $\boldsymbol{p}_{i'-1} \neq \uparrow$.
- The pair of insertion transitions leading to depth $i' \in \{1, \ldots, |\boldsymbol{p}|-1\}$ can be used in step j' for $j' \geq 3$ of the run of the ε-NFA only if there is an active non-auxiliary state in depth i' and step $j'-2$ of the run of the ε-NFA and $\boldsymbol{t}_{j'} = \uparrow$ and $\boldsymbol{t}_{j'-1} \neq \uparrow$.

Moreover, since we are interested only in the topmost active non-auxiliary state in depth i' and step j' of the run of the ε-NFA, we can focus only on those transitions whose source is the topmost active state in the corresponding depth and step of the run of the ε-NFA. If those transitions yield different states, we choose the one with the lowest level. During iteration i', the algorithm simulates exactly the steps mentioned above:

- Step 2a corresponds to checking whether matching or relabeling transition is possible (note that they are exclusive). If $\boldsymbol{p}_{i'} = \boldsymbol{t}_{j'}$, then c_1 is set to $\boldsymbol{D}_{i'-1,j'-1}$. Value $\boldsymbol{D}_{i'-1,j'-1}$ contains the level of the topmost active non-auxiliary state in depth $i'-1$ of the ε-NFA and step $j'-1$ of the run of the ε-NFA. In other words, in depth $i'-1$ of the ε-NFA and step $j'-1$ of the run of the ε-NFA the topmost active non-auxiliary state is $(i'-1)^l$, where $l = \boldsymbol{D}_{i'-1,j'-1}$. Since $\boldsymbol{p}_{i'} = \boldsymbol{t}_{j'}$, we can move to state $(i')^l$ using the match transition. If $\boldsymbol{p}_{i'} \neq \boldsymbol{t}_{j'}$ and $\boldsymbol{p}_{i'}, \boldsymbol{t}_{j'} \neq \uparrow$, then c_1 is set to $\boldsymbol{D}_{i'-1,j'-1}+1$. This corresponds to moving from state $(i'-1)^l$ to state $(i')^{l+1}$ using the relabeling transition. If $c_1 > k$, then no state in depth i' and step j' of the run of the ε-NFA can be reached using a match or relabeling transition since there are only k levels. If $\boldsymbol{p}_{i'} \neq \boldsymbol{t}_{j'}$ and $p_{i'}$ or $t_{j'}$ is equal to the bar symbol, then no state in depth i' and step j' of the run of the ε-NFA can be reached using a match or relabeling transition. This corresponds to setting c_1 to ∞ in the dynamic programming algorithm.
- Step 2b corresponds to checking whether deletion transition is possible. If $i' \geq 3$, $\boldsymbol{p}_{i'} = \uparrow$, and $\boldsymbol{p}_{i'-1} \neq \uparrow$, then c_2 is set to $\boldsymbol{D}_{i'-2,j'}+1$. Value $\boldsymbol{D}_{i'-2,j'}$ contains the level of the topmost active non-auxiliary state in depth $i'-2$ of the ε-NFA and step j' of the run of the ε-NFA. In other words, in depth $i'-2$ of the ε-NFA and step j' of the run of the ε-NFA the topmost active non-auxiliary state is $(i'-2)^l$, where $l = \boldsymbol{D}_{i'-2,j'}$. Since $i' \geq 3$, $\boldsymbol{p}_{i'} = \uparrow$, and $\boldsymbol{p}_{i'-1} \neq \uparrow$, we can move to state $(i')^{l+1}$ using the deletion transition. If $i' < 3$ or $\boldsymbol{p}_{i'} \neq \uparrow$, or $\boldsymbol{p}_{i'-1} = \uparrow$, then no state in depth i' and step j' of the run of the ε-NFA can be reached using a deletion transition. This corresponds to setting c_2 to ∞ in the dynamic programming algorithm.
- Step 2c corresponds to checking whether a pair of insertion transitions is possible. If $j' \geq 3$, $\boldsymbol{t}_{j'} = \uparrow$, $\boldsymbol{t}_{j'-1} \neq \uparrow$, and $i' < |\boldsymbol{p}|$, then c_3 is set to $\boldsymbol{D}_{i',j'-2}+1$. Value $\boldsymbol{D}_{i',j'-2}$ contains the level of the topmost active non-auxiliary state in depth i' of the ε-NFA and step $j'-2$ of the run of the ε-NFA. In other words, in depth i' of the ε-NFA and step $j'-2$ of the run of the ε-NFA the topmost active non-auxiliary state is $(i')^l$, where $l = \boldsymbol{D}_{i',j'-2}$. Since $j' \geq 3$, $\boldsymbol{t}_{j'} = \uparrow$, and $\boldsymbol{t}_{j'-1} \neq \uparrow$, we can move to state $(i')^{l+1}$ using the pair of insertion transition that leads through auxiliary state $(i')_1^{l+1}$. If $j' < 3$ or $\boldsymbol{t}_{j'} \neq \uparrow$ or $\boldsymbol{t}_{j'-1} = \uparrow$, or $i' = |\boldsymbol{p}|$, then no state in depth i' and step j' of the run of the ε-NFA can be reached using a pair of insertion transitions. This corresponds to setting c_3 to ∞ in the dynamic programming algorithm.
- Step 2d executes $\boldsymbol{D}_{i',j'} \leftarrow \mathsf{min}(c_1, c_2, c_3)$. This clearly corresponds saving the level of the topmost active state in depth i' and step j' of the run of the ε-NFA (or saving value greater than k or ∞ if there is no such state).

Thus, before the start of iteration $i'+1$, $\boldsymbol{D}_{i'',j'}$ contains the level of the topmost active non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA for each $i'' \in \{0, \ldots, i'\}$. At termination $i = |\boldsymbol{p}|+1$, value $\boldsymbol{D}_{i'',j'}$ corresponds to the level of the topmost active non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA for each $i'' \in \{0, \ldots, |\boldsymbol{p}|\}$.

Thus, after the execution of the inner loop, we get that for each $i \in \{0, \dots, |\boldsymbol{p}|\}$ and $j'' \in \{0, \dots, j'\}$, value $\boldsymbol{D}_{i,j''}$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA is active. At termination $j = |\boldsymbol{t}| + 1$, value $\boldsymbol{D}_{i,j''}$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA for each $i \in \{0, \dots, |\boldsymbol{p}|\}$ and $j'' \in \{0, \dots, |\boldsymbol{t}|\}$. Hence the dynamic programming simulates the 1-degree matching ε-NFA. ◀

We note that all values $\boldsymbol{D}_{i,j} > k$ in the array can be replaced by the value ∞ representing a number of errors greater than k. This is because such values can no longer produce an occurrence. Furthermore, Step 3 of Algorithm 5.38 can be merged with Step 2 so that we report occurrences as we fill the array instead of waiting until the whole array is filled. We now discuss the time and space complexity of Algorithm 5.38.

▶ **Theorem 5.40** (Time complexity of Algorithm 5.38). *Let $\mathcal{P}$ and $\mathcal{T}$ be ordered labeled trees with m and n nodes, respectively. Given the prefix bar notation of $\mathcal{P}$ and $\mathcal{T}$ and $k \geq 0$, the time complexity of Algorithm 5.38 is $\mathcal{O}(m \cdot n)$.*

Proof. The dynamic programming approach builds a two-dimensional array $\boldsymbol{D}$ of size $(2m+1) \cdot (2n+1)$. The computation of the value at each position of $\boldsymbol{D}$ depends on three positions at most and this computation executes in constant time. ◀

▶ **Theorem 5.41** (Space complexity of Algorithm 5.38). *Let $\mathcal{P}$ and $\mathcal{T}$ be ordered labeled trees with m and n nodes, respectively. Assume $m \leq n$. Given the prefix bar notation of $\mathcal{P}$ and $\mathcal{T}$ and $k \geq 0$, Algorithm 5.38 can be implemented to use $\mathcal{O}(m)$ space.*

Proof. The dynamic programming approach builds a two-dimensional array $\boldsymbol{D}$ of size $(2m + 1) \cdot (2n+1)$. The computation of the value at each position of $\boldsymbol{D}$ depends on three positions at most: $\boldsymbol{D}_{i-1,j-1}$, $\boldsymbol{D}_{i-2,j}$, and $\boldsymbol{D}_{i,j-2}$. Thus, every column can be computed using just two previous columns. Therefore, only $4m$ space is needed in order to compute all values and the space complexity results in $\mathcal{O}(m)$. ◀

As stated earlier, our dynamic programming algorithm is inspired by the algorithm for the inexact string pattern matching under the simple Levenshtein distance. We recall that the dynamic programming algorithm for the inexact string pattern matching is an extension of the algorithm that computes the simple Levenshtein distance between two strings; see Algorithm 4.4. In Section 4.1.1.1, we discussed that this algorithm can be modified so that the match operation is outside the minimum function. However, the analogous modification cannot be used for Algorithm 5.38. This is because, it is not guaranteed that value $\boldsymbol{D}_{i-1,j-1}$ is always smaller than values $\boldsymbol{D}_{i-2,j} + 1$ and $\boldsymbol{D}_{i,j-2} + 1$. See an illustration in Figure 5.7.

5.5 1-degree matching automaton for the simple 1-degree edit distance

In this section, we focus on the NFFOBI tree pattern matching problem under the simple 1-degree edit distance; see Problem 5.13. In contrast to its constrained version, the distance d_{s} is based on the length of a shortest edit script, which always exists for any given pair of trees.

We present an automata-based solution in the same way as we did in the previous section. However, first we show that the 1-degree matching automaton can be constructed as a pushdown automaton since we believe that its structure is more intuitive than the structure of the finite automaton. Then, we show that the PDA can be transformed into an ε-NFA accepting the same language due to its restricted use of the pushdown store. Finally, we briefly discuss the deterministic version of the 1-degree matching finite automaton and focus in detail on a simulation of the 1-degree matching ε-NFA using dynamic programming.

$\boldsymbol{D}$	j	0	1	2	3	4	5	6
i	-	-	c	b	↑	a	↑	↑
0	-	0	0	0	0	0	0	0
1	c	∞	0	1	1	1	2	∞
2	b	∞	∞	0	∞	2	∞	∞
3	↑	∞	1	2	0	2	1	∞
4	↑	∞	∞	∞	2	∞	2	1

Figure 5.7 An example of dynamic programming approach to the NFFOBI tree pattern matching problem under d_s^c that finds all occurrences of pattern tree $\mathcal{P}$ in input tree $\mathcal{T}$ with up to k errors, where $\mathsf{prefBar}(\mathcal{P}) = \mathtt{cb} \uparrow\uparrow$ and $\mathsf{prefBar}(\mathcal{T}) = \mathtt{cb} \uparrow \mathtt{a} \uparrow\uparrow$. For $k = 1$, there is one occurrence of $\mathcal{P}$ in $\mathcal{T}$; the one found at position 6 of $\boldsymbol{t} = \mathsf{prefBar}(\mathcal{T})$. The corresponding edit script consists of one insertion operation illustrated in the table by blue entries. Note that although $\boldsymbol{p}_3 = \boldsymbol{t}_5$, it is more favorable to apply insertion operation than copy value $\boldsymbol{D}_{2,4}$ which indicates the match.

Our algorithm constructing the 1-degree matching automaton for the simple 1-degree edit distance uses the subtree jump table as an auxiliary data structure. We recall that we described this data structure in Section 4.2.2.2. Janoušek et al. [32] presented an algorithm computing the subtree jump table for ordered ranked trees in the prefix notation. Later, Trávníček [33, Algorithm 23] proposed an algorithm constructing the subtree jump table for ordered ranked trees in the prefix *ranked* bar notation in which each node has a precomputed arity (number of children). In this section, we propose an alternative algorithm that works directly with the prefix (unranked) bar notation of trees; see Algorithm 5.42.

The central idea of Algorithm 5.42 is to use a stack to record the labels' positions. When the bar symbol is found, the position of the corresponding label is popped from the stack. Given an ordered labeled tree $\mathcal{P}$, the algorithm constructs the subtree jump table for $\mathsf{prefBar}(\mathcal{P})$ in time linear to the length of $\mathsf{prefBar}(\mathcal{P})$.

▶ **Algorithm 5.42** (Computation of the subtree jump table for an ordered labeled tree in the prefix bar notation)**.** Let $\mathcal{P}$ be an ordered labeled tree. Given $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$, the algorithm constructs the subtree jump table for $\boldsymbol{p}$ which is represented as a one-dimensional array $\boldsymbol{S^p}$ with ranges 1 to $|\boldsymbol{p}|$. The principal internal data structure is a stack $\boldsymbol{Y}$ with operations push that inserts a symbol on its top and pop that returns and deletes the top symbol.

1. (Initialize data structures.)
   a. Create an empty stack $\boldsymbol{Y}$.
   b. Create an array $\boldsymbol{S^p}$ of size $|\boldsymbol{p}|$.
2. (Fill array.) For each position i in $\boldsymbol{p}$:
   a. If $\boldsymbol{p}_i \neq \uparrow$, then $\mathsf{push}(i)$ into $\boldsymbol{Y}$.
   b. Otherwise, set $\boldsymbol{S}^{\boldsymbol{p}}_i \leftarrow \mathsf{pop}(\boldsymbol{Y}) - 1$ and set $\boldsymbol{S}^{\boldsymbol{p}}_{i'} \leftarrow i + 1$, where $i' = \boldsymbol{S}^{\boldsymbol{p}}_i + 1$.
3. (Return.) Return $\boldsymbol{S^p}$.

▶ **Lemma 5.43** (Correctness of Algorithm 5.42)**.** *Let $\mathcal{P}$ be an ordered labeled tree. Given* $\mathsf{prefBar}(\mathcal{P})$*, Algorithm 5.42 constructs the subtree jump table for* $\mathsf{prefBar}(\mathcal{P})$*.*

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. To prove that $\boldsymbol{S^p}$ is the subtree jump table for $\boldsymbol{p}$, we need to show that for each position i in $\boldsymbol{p}$ the following holds: If $\boldsymbol{p}_i \in \Sigma$, then $\boldsymbol{S}^{\boldsymbol{p}}_i = j$ such that j is the first position in $\boldsymbol{p}$ after the bar symbol corresponding to $\boldsymbol{p}_i$; and if $\boldsymbol{p}_i = \uparrow$, then $\boldsymbol{S}^{\boldsymbol{p}}_i = j$ such that j is the first position in $\boldsymbol{p}$ before the label corresponding to $\boldsymbol{p}_i$.

The algorithm reads $\boldsymbol{p}$ from left to right, and uses stack $\boldsymbol{Y}$ to save each position i, where $\boldsymbol{p}_i \in \Sigma$. Therefore, when the bar symbol symbol is encountered, its corresponding label position is at the top of $\boldsymbol{Y}$. The algorithm retrieves it and subtracts one (to retrieve the preceding position in $\boldsymbol{p}$). Thus, $\boldsymbol{S}^{\boldsymbol{p}}_i$ contains correct values if $\boldsymbol{p}_i = \uparrow$.

The value i' is the original number popped from $\boldsymbol{Y}$. Therefore, i' is the label position that corresponds to the bar symbol at position i. Moreover, value $i+1$ points to the first position after the bar symbol. Thus, $\boldsymbol{S}^{\boldsymbol{p}}_i$ contains correct values if $\boldsymbol{p}_i \in \Sigma$. ◀

5.5.1 Pushdown automaton

We introduce the algorithm that constructs the 1-degree matching automaton for d_{s} in the same was as we did in the case of the 1-degree matching automaton for $d^{\mathrm{c}}_{\mathrm{s}}$. First, we construct an automaton accepting the pattern dictionary for a given pattern tree, maximum number of errors, and distance d_{s}. Then, we extend it by adding loop transitions to its start state to enable the automaton to read the prefix bar notation of any input tree.

As stated earlier, we first construct the 1-degree matching automaton as a pushdown automaton, see Algorithms 5.44 and 5.46. The transitions for match and relabeling operations are constructed similarly as we explained in the previous section. The difference is in the construction of the insertion and deletion transitions that are now allowed to insert or delete a bottom-up subtree of any size up to the maximum number of errors. Insertion operations are created using the pushdown store which is used to match label-bar pairs. Deletion transitions use the subtree jump table constructed for the prefix bar notation of a given pattern tree.

We show an example of the PDA constructed by Algorithm 5.44 and the 1-degree matching PDA constructed by Algorithm 5.46 for $\boldsymbol{p} = \mathtt{ab} \uparrow \mathtt{b} \uparrow \mathtt{ac} \uparrow\uparrow\uparrow$ and $k = 2$ in Figure 5.8. Insertion operations are represented by *vertical* transitions and loop transitions in states i^l, where $i \in \{1, \ldots, |\boldsymbol{p}| - 1\}$ and $l \in \{1, \ldots, k\}$; *diagonal* ε-transitions represent deletion operations, and the remaining *diagonal* transitions represent relabeling operations.

The pushdown automata constructed by Algorithms 5.44 and 5.46 have $1 + |\boldsymbol{p}| \cdot (k+1)$ states and can be built in time $\mathcal{O}(k \cdot |\boldsymbol{p}|)$, where $\boldsymbol{p}$ is the prefix bar notation of a given pattern tree and k represents the maximum number of errors allowed.

▶ **Algorithm 5.44** (Construction of a PDA accepting the pattern dictionary for d_{s})**.** Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Let $k \geq 0$. Given $\boldsymbol{p}$ and k, the algorithm constructs a PDA $\mathcal{M} = (Q, \Sigma_{\uparrow}, \Gamma, \delta, q_0, \bot, F)$ such that $L(\mathcal{M}) = L(\boldsymbol{p}, k, d_{\mathrm{s}})$.

1. (Compute the subtree jump table for $\boldsymbol{p}$.) Compute the subtree jump table $\boldsymbol{S}^{\boldsymbol{p}}$ for $\boldsymbol{p}$ using Algorithm 5.42.

2. (Define the set of states.) Each state is labeled by i^l, where $i \in \{0, \ldots, |\boldsymbol{p}|\}$ and $l \in \{0, \ldots, k\}$. Specifically, states are labeled so they can be arranged into rows and columns similarly as illustrated in Figure 5.3. The difference is that auxiliary states are not used this time. We say that a state labeled by i^l is at *depth* i and at *level* l. There are $(k+1)$ rows, one for each level. Each row, except the one for level 0, is composed of $|\boldsymbol{p}|$ states of depths $1, 2, \ldots, |\boldsymbol{p}|$. In the row for level 0, there are $|\boldsymbol{p}| + 1$ states of depths $0, 1, 2, \ldots, |\boldsymbol{p}|$.

$$\text{Set } Q \leftarrow \{0^0\} \cup \{i^l : 1 \leq i \leq |\boldsymbol{p}| \land 0 \leq l \leq k\}.$$

3. (Define the start state.) Set $q_0 \leftarrow 0^0$.

4. (Define the set of final states.) Every state at depth $|\boldsymbol{p}|$ is final. That is, every row ends in one of the final states.

$$\text{Set } F \leftarrow \{|\boldsymbol{p}|^l : 0 \leq l \leq k\}.$$

5. (Initialize pushdown alphabet.) Set $\Gamma = \{\mathtt{s}, \perp\}$.
6. (Add transitions indicating match.)
   a. Set $\delta\left(0^0, \boldsymbol{p}_1, \perp\right) \leftarrow \left\{\left(1^0, \perp\right)\right\}$.
   b. For each position $i \in \{2, \ldots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \ldots, k\}$, set $\delta\left((i-1)^l, \boldsymbol{p}_i, \perp\right) \leftarrow \left\{\left(i^l, \perp\right)\right\}$.
7. (Add transitions for relabeling operations.)
   a. If $k \geq 1$, then for each $a \in \Sigma \setminus \{\boldsymbol{p}_1\}$, set $\delta\left(0^0, a, \perp\right) \leftarrow \left\{\left(1^1, \perp\right)\right\}$.
   b. For each position $i \in \{2, \ldots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \ldots, k-1\}$: If $\boldsymbol{p}_i \neq \uparrow$, then for each $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$, set $\delta\left((i-1)^l, a, \perp\right) \leftarrow \left\{\left(i^{l+1}, \perp\right)\right\}$.
8. (Add transitions for insertion operations.) Two types of transitions represent insertion operations: insertion of a symbol $a \in \Sigma$ and the corresponding bar symbol. Since we can insert a bottom-up subtree of any size smaller than or equal to k, the corresponding bar symbol does not need to follow the symbol immediately. Thus, for each $a \in \Sigma$, we push the pushdown symbol $\mathtt{s}$ to the pushdown store, and for each bar symbol, one pushdown symbol $\mathtt{s}$ is popped. Hence, by reading a symbol $a \in \Sigma$, we start inserting a bottom-up subtree with the root labeled by a, and the subtree insertion is complete as soon as the pushdown store does not contain any symbols $\mathtt{s}$. Note that another insertion can immediately occur once the insertion of a bottom-up subtree is complete.

   For each position $i \in \{2, \ldots, |\boldsymbol{p}|\}$, number of errors $l \in \{0, \ldots, k-1\}$, and symbol $a \in \Sigma$:
   a. Set $\delta\left((i-1)^l, a, \varepsilon\right) \leftarrow \left\{\left((i-1)^{l+1}, \mathtt{s}\right)\right\}$.
   b. Set $\delta\left((i-1)^{l+1}, \uparrow, \mathtt{s}\right) \leftarrow \left\{\left((i-1)^{l+1}, \varepsilon\right)\right\}$.
9. (Add transitions for deletion operations.) Deletion operations are represented by ε-transitions. These transitions allow deletion of any bottom-up subtree whose size is smaller than or equal to k, and are constructed using the subtree jump table $\boldsymbol{S}^{\boldsymbol{p}}$ and its properties shown in Remark 4.20.

   For each position $i \in \{2, \ldots, |\boldsymbol{p}|-2\}$ and number of errors $l \in \{0, \ldots, k-1\}$: If $\boldsymbol{p}_i \in \Sigma$ and $l + \mathit{size} \leq k$, where $\mathit{size} = (\boldsymbol{S}^{\boldsymbol{p}}_i - i)/2$, then the proper bottom-up subtree of $\boldsymbol{p}$ that starts at position i can be deleted. This can be done by setting

   $$\delta\left((i-1)^l, \varepsilon, \perp\right) \leftarrow \left\{\left((\boldsymbol{S}^{\boldsymbol{p}}_i - 1)^{l+\mathit{size}}, \perp\right)\right\}.$$
10. (Return.) Return $\mathcal{M} = (Q, \Sigma_\uparrow, \Gamma, \delta, q_0, \perp, F)$.

▶ **Lemma 5.45** (Correctness of Algorithm 5.44). *Given an ordered labeled tree $\mathcal{P}$ and $k \geq 0$, Algorithm 5.44 constructs a PDA $\mathcal{M} = (Q, \Sigma_\uparrow, \Gamma, \delta, q_0, \perp, F)$ such that $L(\mathcal{M}) = L(\mathsf{prefBar}(\mathcal{P}), k, d_\mathrm{s})$.*

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. The proof falls naturally into two parts.

($\subseteq$) Assume $\boldsymbol{x} \in L(\mathcal{M})$. Thus, there exists $l \in \{0, \ldots, k\}$ such that $\left(0^0, \boldsymbol{x}, \perp\right) \vdash^*_{\mathcal{M}} \left(|\boldsymbol{p}|^l, \varepsilon, \alpha\right)$, where $\alpha \in \Gamma^*$ and there is no $l' < l$ such that $\left(0^0, \boldsymbol{x}, \perp\right) \vdash^*_{\mathcal{M}} \left(|\boldsymbol{p}|^{l'}, \varepsilon, \beta\right)$, where $\beta \in \Gamma^*$. It is easy to check that it always holds that $\alpha = \beta = \perp$. We show that $\boldsymbol{x} \in L(\boldsymbol{p}, k, d_\mathrm{s})$ by showing that $d_\mathrm{s}(\boldsymbol{p}, \boldsymbol{x}) = l$. We use induction on the maximum number of errors k.

- Assume $k = 0$. Then, $\left(0^0, \boldsymbol{x}, \perp\right) \vdash^*_{\mathcal{M}} \left(|\boldsymbol{p}|^0, \varepsilon, \perp\right)$. Thus, $\mathcal{M}$ reads $\boldsymbol{x}$ using transitions created in Step 6. Therefore, $\boldsymbol{x} = \boldsymbol{p}$, and it follows that $d_\mathrm{s}(\boldsymbol{p}, \boldsymbol{x}) = 0$.

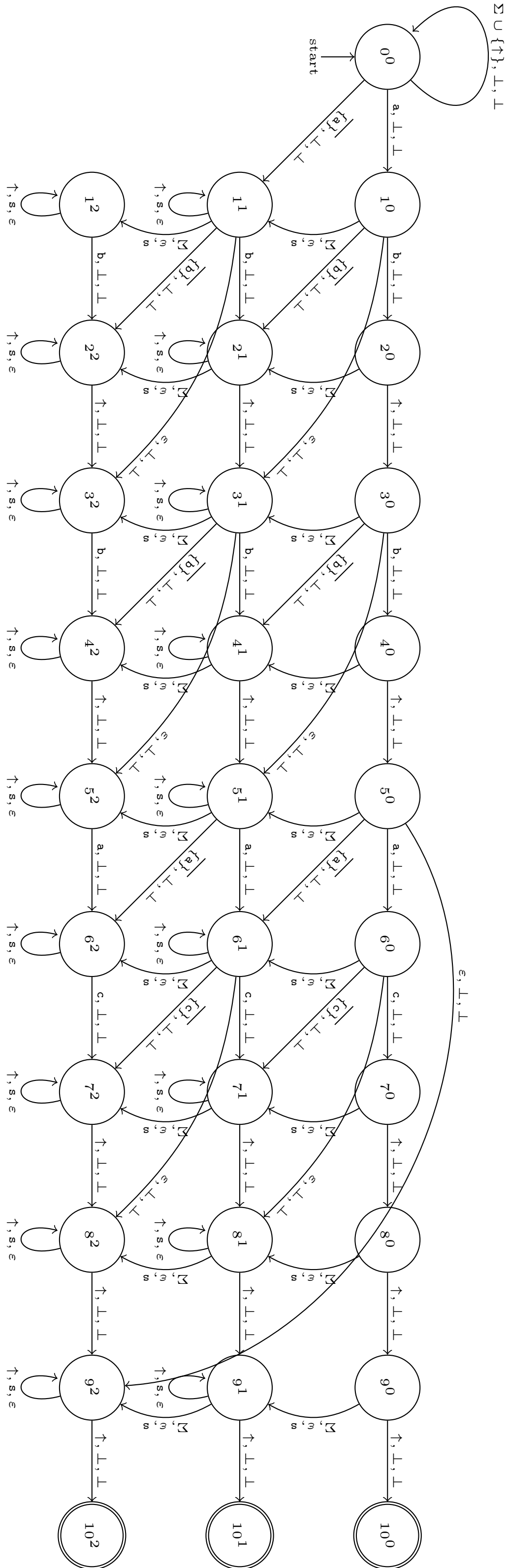


Figure 5.8 The PDA constructed by Algorithm 5.44 accepting the pattern dictionary or the 1-degree matching PDA constructed by Algorithm 5.46 for distance d_s, maximum number of errors $k = 2$, and pattern tree $\mathcal{P}$ illustrated in Figure 5.1a. If the automaton contains loop transitions in the start state, it is the 1-degree matching PDA $\mathcal{M}(\mathsf{prefBar}(\mathcal{P}), k, d_s)$. If the loop transitions in the start state are removed, it is the PDA accepting the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d_s)$.

- Assume $k \geq 1$ and that the claim holds for errors smaller than k. For each $\boldsymbol{x} \in L(\mathcal{M})$, there exists $l \in \{0, \ldots, k\}$ such that $\left(0^0, \boldsymbol{x}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^l, \varepsilon, \perp\right)$ and there is no $l' < l$ such that $\left(0^0, \boldsymbol{x}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^{l'}, \varepsilon, \perp\right)$. For $l \in \{0, \ldots, k-1\}$, we get that $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{x}) = l$ by the induction hypothesis. Let $\boldsymbol{x'}$ be a string such that $\left(0^0, \boldsymbol{x'}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^k, \varepsilon, \perp\right)$, where there is no $k' < k$ such that $\left(0^0, \boldsymbol{x'}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^{k'}, \varepsilon, \perp\right)$. Our goal is to show that it follows that $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{x'}) = k$.

  If $\left(0^0, \boldsymbol{x'}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^k, \varepsilon, \perp\right)$, where there is no $k' < k$ such that $\left(0^0, \boldsymbol{x'}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^{k'}, \varepsilon, \perp\right)$, then $\mathcal{M}$ must use (at some point) a transition created in Steps 7, 8, or 9 to read $\boldsymbol{x'}$. Let $\boldsymbol{z'}$ be the longest suffix of $\boldsymbol{x'}$ such that that $\mathcal{M}$ reads it on level k using only transitions created in Step 6. It is easy to check that this sequence of transitions corresponds to reading a nonempty proper suffix of $\boldsymbol{p}$, and that the source state of the transition reading the first symbol of $\boldsymbol{z'}$ is state $(|\boldsymbol{p}| - |\boldsymbol{z'}|)^k$. Automaton $\mathcal{M}$ moves to state $(|\boldsymbol{p}| - |\boldsymbol{z'}|)^k$ using one of the following ways:

  - using a transition created in Step 7,
  - using a transition created in Step 9, or
  - using a sequence of transitions created in Step 8 so that the pushdown store contains only one symbol (which is the start symbol) immediately before the sequence is initiated and after it is executed.

  The source state of this transition (or a sequence of transitions) is state q^l, where $q \in \{0, \ldots, |\boldsymbol{p}| - |\boldsymbol{z'}|\}$ and $l \in \{0, \ldots, k-1\}$. Thus, we can divide $\boldsymbol{x'}$ into three parts as follows: $\boldsymbol{x'} = \boldsymbol{yuz'}$, where $\boldsymbol{z'}$ is the longest suffix of $\boldsymbol{x'}$ such that $\mathcal{M}$ reads it on level k using only transitions created in Step 6, $\boldsymbol{u}$ is the immediately preciding part of $\boldsymbol{x'}$ that is read while $\mathcal{M}$ moves from q^l to $(|\boldsymbol{p}| - |\boldsymbol{z'}|)^k$, and $\boldsymbol{y}$ is the remaining prefix of $\boldsymbol{x'}$ that moves $\mathcal{M}$ from 0^0 to q^l. From q^l, we can use transitions created in Step 6 and move to the final state $|\boldsymbol{p}|^l$. It easy to check that this sequence of transitions corresponds to reading a nonempty suffix $\boldsymbol{z}$ of $\boldsymbol{p}$. Thus, $\left(0^0, \boldsymbol{yz}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^l, \varepsilon, \perp\right)$. Moreover, it can be shown that there cannot be $l' < l$ such that $\left(0^0, \boldsymbol{yz}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^{l'}, \varepsilon, \perp\right)$; otherwise, there would exist $k' < k$ such that $\left(0^0, \boldsymbol{x'}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^{k'}, \varepsilon, \perp\right)$. Therefore, we get that $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{yz}) = l$ by the induction hypothesis. Since $\boldsymbol{p}$ can be written as $\boldsymbol{p} = \boldsymbol{wz}$, it follows that each of the l edit operations were applied to $\boldsymbol{w}$ in order to change it to $\boldsymbol{y}$. Thus, in order to prove that $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{x'}) = k$, we need to show that reading $\boldsymbol{uz'}$ corresponds to applying $k - l$ edit operations on $\boldsymbol{z}$ (recall that $\boldsymbol{x'} = \boldsymbol{yuz'}$).

  It is easy to see that reading $\boldsymbol{z'}$ corresponds to reading suffix of $\boldsymbol{z}$. Reading $\boldsymbol{u}$ corresponds to reading one of the following strings: string of length one, ε, or a string from $L(\Sigma, \uparrow)$. Reading string of length one corresponds to the situation when $k - l = 1$ and when $\mathcal{M}$ uses a transition created in Step 7 to move from q^{k-1} to $(|\boldsymbol{p}| - |\boldsymbol{z'}|)^k$, where $q = |\boldsymbol{p}| - |\boldsymbol{z'}| - 1$. This corresponds to applying relabeling operation on the first symbol of $\boldsymbol{z}$. Reading ε corresponds to the situation when $\mathcal{M}$ uses a transition created in Step 9 to move from q^l to $(|\boldsymbol{p}| - |\boldsymbol{z'}|)^k$, where $q = |\boldsymbol{p}| - |\boldsymbol{z'}| - 2(k-l)$. This corresponds to using $k - l$ deletion operations on $\boldsymbol{z}$ which result into deleting the prefix of $\boldsymbol{z}$ of length $2(k-l)$ representing a bottom-up subtree of $\mathcal{P}$ of size $k - l$. Finally, reading a string from $L(\Sigma, \uparrow)$ corresponds to the situation when $\mathcal{M}$ uses a sequence of transitions created in Step 8 so that the pushdown store contains only one symbol (which is the start symbol) immediately before the sequence is initiated and after it is executed. This moves $\mathcal{M}$ from q^l to $(|\boldsymbol{p}| - |\boldsymbol{z'}|)^k$, where $q = |\boldsymbol{p}| - |\boldsymbol{z'}|$, and corresponds to using $k - l$ insertion operations on $\boldsymbol{z}$ which result into inserting the prefix to $\boldsymbol{z}$ of length $2(k-l)$ representing a tree of size $k - l$. Thus, reading $\boldsymbol{uz'}$ corresponds to applying $k - l$ edit operations on $\boldsymbol{z}$. It follows that $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{x}) = k$.

($\supseteq$) Assume $\boldsymbol{x} \in L(\boldsymbol{p}, k, d_{\mathrm{s}})$. Thus, there exists $l \in \{0, \ldots, k\}$ such that $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{x}) = l$. We show that $\boldsymbol{x} \in L(\mathcal{M})$ by showing that $\left(0^0, \boldsymbol{x}, \perp\right) \vdash_{\mathcal{M}}^{*} \left(|\boldsymbol{p}|^l, \varepsilon, \alpha\right)$, where $\alpha \in \Gamma^*$. We use induction on the maximum number of errors k.

- Assume $k = 0$. Then, $L(\boldsymbol{p}, k, d_{\mathrm{s}}) = \{\boldsymbol{p}\}$. From Step 6 of the algorithm, it follows that $\left(0^0, \boldsymbol{p}, \bot\right) \vdash^*_{\mathcal{M}} \left(|\boldsymbol{p}|^0, \varepsilon, \bot\right)$. Thus, $\boldsymbol{p} \in L(\mathcal{M})$.

- Assume $k \geq 1$ and that the claim holds for errors smaller than k. Then, $L(\boldsymbol{p}, k, d_{\mathrm{s}}) = L(\boldsymbol{p}, k-1, d_{\mathrm{s}}) \cup \{\boldsymbol{y} : \boldsymbol{y} \in L(\Sigma, \uparrow) \wedge d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{y}) = k\}$. By the induction hypothesis, we have that $L(\boldsymbol{p}, k-1, d_{\mathrm{s}}) \subseteq L(\mathcal{M})$. Our goal is to show that $\{\boldsymbol{y} : \boldsymbol{y} \in L(\Sigma, \uparrow) \wedge d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{y}) = k\} \subseteq L(\mathcal{M})$. We know that for each such $\boldsymbol{y}$, there is an edit script between $\boldsymbol{p}$ and $\boldsymbol{y}$ with k operations. If there are multiple (leaf) deletion operations in the edit script that result in deleting a bottom-up subtree of $\mathcal{P}$, they can be represented by one (subtree) deletion operation whose cost is the number of (leaf) deletion operations it represents. The similar representation can be used for potential multiple (leaf) insertion operations that inserts a new bottom-up subtree into $\mathcal{P}$. The relabeling operations, (subtree) deletion operations, and (subtree) insertion operations in the edit script can be sorted from left to right according to which node in $\mathcal{P}$ they refer to (a subtree deletion operation refers to the root of the subtree that is being deleted, a subtree insertion operations refers to a node in $\mathcal{P}$ that is the parent of the root of the subtree that is being inserted). This way we can transform $\boldsymbol{p}$ to $\boldsymbol{y}$ by traversing $\boldsymbol{p}$ from its beginning to its end.

  If we remove the rightmost operation from such a script, then we clearly transform $\boldsymbol{p}$ to string $\boldsymbol{y'}$, where $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{y'}) = l$ such that $l \in \{0, \ldots, k-1\}$. For each such $\boldsymbol{y'}$, we have that $\left(0^0, \boldsymbol{y'}, \bot\right) \vdash^*_{\mathcal{M}} \left(|\boldsymbol{p}|^l, \varepsilon, \alpha\right)$, where $\alpha \in \Gamma^*$ by the induction hypothesis. It is easy to check that it always holds that $\alpha = \bot$, and that there is a nonempty suffix $\boldsymbol{z}$ of $\boldsymbol{p}$ that is intact (not changed by any of the remaining operations in the edit script). The intact suffix $\boldsymbol{z}$ is read using transitions created in Step 6. Now we apply on $\boldsymbol{z}$ the operation that has been removed. This operation can be one of the following:

  - The relabeling operation in which case $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{y'}) = k - 1$ and thus the suffix $\boldsymbol{z}$ is read using transitions created in Step 6 between states on level $k-1$. Applying the relabeling operation to $\boldsymbol{p}_i$ corresponds to using a transition that increases level by one and lead from a state at depth $i-1$ to a state at depth i. These transitions are created in Step 7 of the algorithm.
  - The (subtree) deletion operation in which case $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{y'}) = k - \mathit{size}$, where size is the size of the subtree that is being deleted. Moreover, it follows that $\boldsymbol{z}$ is read using transitions created in Step 6 between states on level $k - \mathit{size}$. Let $\boldsymbol{p}_{i\ldots j}$ represent the substring of $\boldsymbol{z}$ that is being deleted by the operation. Applying the (subtree) deletion operation corresponds to using an ε-transition that increases level by size and lead from a state at depth $(i-1)$ to a state at depth j. Value j corresponds to the value $\boldsymbol{S}^{\boldsymbol{p}}_i - 1$, where $\boldsymbol{S}^{\boldsymbol{p}}$ is the subtree jump table for $\boldsymbol{p}$. These transitions are created in Step 9 of the algorithm.
  - The (subtree) insertion operation in which case $d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{y'}) = k - \mathit{size}$, where size is the size of the subtree that is being inserted. Moreover, it follows that $\boldsymbol{z}$ is read using transitions created in Step 6 between states on level $k - \mathit{size}$. Applying the (subtree) insertion operation corresponds to using a sequence of transitions that reads a string $\boldsymbol{w} \in L(\Sigma, \uparrow)$ of length $2 \cdot \mathit{size}$. This sequence of transitions increases level by size and leads between states at depth $i-1$, where $i \in \{2, \ldots, |\boldsymbol{p}|\}$ is the position in $\boldsymbol{p}$ where the new subtree is being inserted. These transitions are created in Step 8 of the algorithm. The automaton verifies that $\boldsymbol{w} \in L(\Sigma, \uparrow)$ by using the pushdown store: for each $a \in \Sigma$, it pushes the pushdown symbol $\mathtt{s}$ to the pushdown store, and for each bar symbol, one pushdown symbol is popped. No other transition can be used until all symbols $\mathtt{s}$ are popped.

  After using the transition (or the sequence of transitions) that corresponds to the rightmost operation in the edit script, the automaton reads the rest of $\boldsymbol{z}$ using transitions created in Step 6 and accepts it by the final state $|\boldsymbol{p}|^k$. Thus, $\left(0^0, \boldsymbol{y}, \bot\right) \vdash^*_{\mathcal{M}} \left(|\boldsymbol{p}|^k, \varepsilon, \bot\right)$ for each $\boldsymbol{y}$ in $\{\boldsymbol{y} : \boldsymbol{y} \in L(\Sigma, \uparrow) \wedge d_{\mathrm{s}}(\boldsymbol{p}, \boldsymbol{y}) = k\}$. Hence, the claim holds. ◀

▶ **Algorithm 5.46** (Construction of the 1-degree matching PDA for d_s). Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $k \geq 0$. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Given $\boldsymbol{p}$ and k, the algorithm constructs the 1-degree matching PDA for $\boldsymbol{p}, k$, and d_s that accepts the language $\{\boldsymbol{x}\boldsymbol{p}' : \boldsymbol{x} \in \Sigma_\uparrow^* \wedge \boldsymbol{p}' \in L(\boldsymbol{p}, k, d_\mathrm{s})\}$ by final state.

1. (Construct automaton for the pattern dictionary.) Construct a PDA $(Q, \Sigma_\uparrow, \Gamma, \delta, q_0, \bot, F)$ accepting the pattern dictionary $L(\boldsymbol{p}, k, d_\mathrm{s})$ by final state using Algorithm 5.44.
2. (Add loop to the start state.) For each symbol $a \in \Sigma_\uparrow$, set $\delta(q_0, a, \bot) \leftarrow \delta(q_0, a, \bot) \cup \{(q_0, \bot)\}$.
3. Return $\mathcal{M} = (Q, \Sigma_\uparrow, \Gamma, \delta, q_0, \bot, F)$.

▶ **Theorem 5.47** (Correctness of Algorithm 5.46). *Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $k \geq 0$. Given* $\mathsf{prefBar}(\boldsymbol{p})$ *and k, Algorithm 5.46 constructs the 1-degree matching PDA for $\boldsymbol{p}, k$, and d_s.*

Proof. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Let $\mathcal{M}$ be the automaton that is returned by the algorithm in Step 3. We show that $L(\mathcal{M}) = \{\boldsymbol{x}\boldsymbol{p}' : \boldsymbol{x} \in \Sigma_\uparrow^* \wedge \boldsymbol{p}' \in L(\boldsymbol{p}, k, d_\mathrm{s})\}$. In Step 1, we create a PDA accepting language $\{\boldsymbol{p}' : \boldsymbol{p}' \in L(\boldsymbol{p}, k, d_\mathrm{s})\}$ by final state. The correctness of this automaton is given by Lemma 5.45. In Step 2, we add a loop transition to its start state for each symbol $a \in \Sigma_\uparrow$. These loop transitions do not change the content of the pushdown store. Thus, the automaton can read arbitrary many symbols and stay at the start state. At any point, the nondeterminism allows the automaton to proceed to another state and read string $\boldsymbol{p}'$ such that $d_\mathrm{s}(\boldsymbol{p}, \boldsymbol{p}') \leq k$. This automaton is returned in Step 3 as automaton $\mathcal{M}$. Therefore, $L(\mathcal{M}) = \{\boldsymbol{x}\boldsymbol{p}' : \boldsymbol{x} \in \Sigma_\uparrow^* \wedge \boldsymbol{p}' \in L(\boldsymbol{p}, k, d_\mathrm{s})\}$. ◀

5.5.2 Finite automaton with ε-transitions

We can observe that the 1-degree matching PDA for d_s uses its pushdown store in a restricted way. First, only two pushdown symbols are used: the start symbol and symbol $\mathtt{s}$. Second, only insertion operations change the content of the pushdown store. Third, since the sum of costs of edit operations is limited by k, the size of the pushdown store is also bounded by k. In other words, the pushdown store serves as a bounded counter. Therefore, we can represent each possible content of the pushdown store by a state instead and construct the 1-degree matching automaton for d_s as a finite automaton.

We present a construction of the 1-degree matching ε-NFA for d_s again in two parts. First, we construct an ε-NFA accepting the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d_\mathrm{s}^\mathrm{c})$ for a given pattern tree $\mathcal{P}$ and a maximum number of errors k; see Algorithm 5.48. This algorithm reuses the structure of the PDA constructed by Algorithm 5.44. The difference is that we use auxiliary states for insertion operations instead of the pushdown store. Then, we can construct the 1-degree matching ε-NFA by extending the automaton built by Algorithm 5.48 by adding a loop transition for every symbol $a \in \Sigma_\uparrow$ to its start state. For brevity, we do not explicitly present such an algorithm since it is almost identical to Algorithm 5.30. The only difference is that we use Algorithm 5.48 in the first step instead of Algorithm 5.28. We show an example of the ε-NFA constructed by Algorithm 5.48 and its corresponding 1-degree matching ε-NFA in Figure 5.10. Both automata have $1 + (k+1) \cdot |\boldsymbol{p}| + (|\boldsymbol{p}| - 1) \cdot \frac{k(k+1)}{2} = \Theta(k^2 \cdot |\boldsymbol{p}|)$ states and can be built in time $\mathcal{O}(k^2 \cdot |\boldsymbol{p}|)$.

▶ **Algorithm 5.48** (Construction of an ε-NFA accepting the pattern dictionary for d_s). Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Let $k \geq 0$. Given $\boldsymbol{p}$ and k, the algorithm constructs an ε-NFA $\mathcal{M} = (Q, \Sigma_\uparrow, \delta, q_0, F)$ accepting language $L(\boldsymbol{p}, k, d_\mathrm{s})$.

1. (Compute the subtree jump table for $\boldsymbol{p}$.) Compute the subtree jump table $\boldsymbol{S}^{\boldsymbol{p}}$ for $\boldsymbol{p}$ using Algorithm 5.42.
2. (Define the set of states.) Each state is labeled by i^l or i^l_s, where $i \in \{0, \dots, |\boldsymbol{p}|\}$, $l \in \{0, \dots, k\}$ and $s \in \{1, \dots, k\}$. We say that a state labeled by i^l or i^l_s is at *depth* i and at *level* l; states

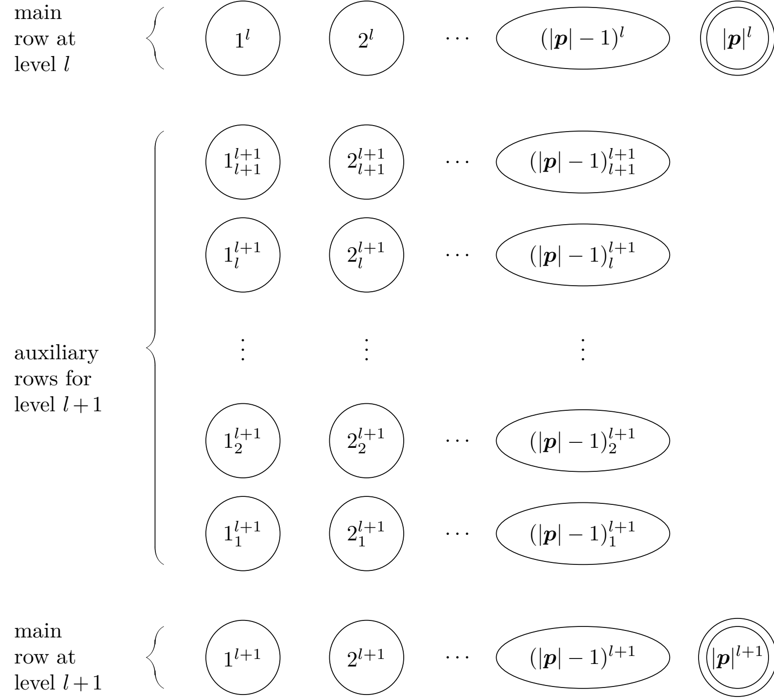


Figure 5.9 The regular structure of the ε-NFA constructed by Algorithm 5.48. Given $k \geq 0$ and a pattern string $\boldsymbol{p} \in L(\Sigma, \uparrow)$, the set of states can be arranged into rows and columns. There are $(k+1)$ main rows, one for each level $l \in \{0, \ldots, k\}$; the figure illustrates two main rows at levels $0 < l < k$ and $l+1$. Moreover, there are $l+1$ auxiliary rows between every two main rows at levels $0 \leq l < k$ and $l+1$. Each main row, except the one for level 0, is composed of $|\boldsymbol{p}|$ states of depths $1, 2, \ldots, |\boldsymbol{p}|$. In the row for level 0, there are $|\boldsymbol{p}|+1$ states of depths $0, 1, 2, \ldots, |\boldsymbol{p}|$. Each auxiliary row is composed of $|\boldsymbol{p}|-1$ states of depths $1, 2, \ldots, |\boldsymbol{p}|-1$.

labeled by i^l_s are called *auxiliary*. Specifically, states are labeled so they can be arranged into rows and columns similarly as illustrated in Figure 5.3. The difference is that there are more auxiliary states this time. In the case of $d^{\mathrm{c}}_{\mathrm{s}}$, there was one auxiliary row between each two consecutive main rows. This time, there are $l+1$ auxiliary rows between main rows at level l and $l+1$; see Figure 5.9. Thus, there are $1+2+3+\cdots+k$ auxiliary rows in the automaton in total.

$$\text{Set } Q \leftarrow \left\{0^0\right\} \cup \left\{i^l : 1 \leq i \leq |\boldsymbol{p}| \wedge 0 \leq l \leq k\right\} \cup \underbrace{\left\{i^l_s : 1 \leq i \leq |\boldsymbol{p}|-1 \wedge 1 \leq s \leq l \leq k\right\}}_{\text{auxiliary states}}.$$

3. (Define the start state.) Set $q_0 \leftarrow 0^0$.

4. (Define the set of final states.) Every state at depth $|\boldsymbol{p}|$ is final. That is, every main row ends in one of the final states.

$$\text{Set } F \leftarrow \left\{|\boldsymbol{p}|^l : 0 \leq l \leq k\right\}.$$

5. (Add transitions indicating match.) Match transitions are created similarly as in the case of the PDA constructed by Algorithm 5.44. The only difference is that we do not use the pushdown store.

a. Set $\delta\left(0^0, \boldsymbol{p}_1\right) \leftarrow \left\{1^0\right\}$.

b. For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \dots, k\}$, set $\delta\left((i-1)^l, \boldsymbol{p}_i\right) \leftarrow \left\{i^l\right\}$.

6. (Add transitions for relabeling operations.) Transitions for relabeling operations are created similarly as in the case of the PDA constructed by Algorithm 5.44. The only difference is that we do not use the pushdown store.

a. If $k \geq 1$, then for each $a \in \Sigma \setminus \{\boldsymbol{p}_1\}$, set $\delta\left(0^0, a\right) \leftarrow \left\{1^1\right\}$.

b. For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \dots, k-1\}$: If $\boldsymbol{p}_i \neq \uparrow$, then for each $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$, set $\delta\left((i-1)^l, a\right) \leftarrow \left\{i^{l+1}\right\}$.

7. (Add transitions for insertion operations.) Similarly, as in the PDA constructed by Algorithm 5.44, two types of transitions represent insertion operations: insertion of a symbol $a \in \Sigma$ and the corresponding bar symbol. However, instead of pushing the symbol $\mathtt{s}$ to the pushdown store for each symbol $a \in \Sigma$, we use auxiliary states for counting. An auxiliary state i_s^l represents a situation in which we need to read s bar symbols to complete the insertion of a subtree that is currently being inserted.

a. (Add transitions from/into states at main rows.) For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$, number of errors $l \in \{0, \dots, k-1\}$, and symbol $a \in \Sigma$:

i. Set $\delta\left((i-1)^l, a\right) \leftarrow \delta\left((i-1)^l, a\right) \cup \left\{(i-1)_1^{l+1}\right\}$.

ii. Set $\delta\left((i-1)_1^{l+1}, \uparrow\right) \leftarrow \left\{(i-1)^{l+1}\right\}$.

b. (Add transitions between auxiliary states.) For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$, number of errors $l \in \{1, \dots, k-1\}$, value $s \in \{1, \dots, l\}$, and symbol $a \in \Sigma$:

i. Set $\delta\left((i-1)_s^l, a\right) \leftarrow \left\{(i-1)_{s+1}^{l+1}\right\}$.

ii. Set $\delta\left((i-1)_{s+1}^{l+1}, \uparrow\right) \leftarrow \left\{(i-1)_s^{l+1}\right\}$.

8. (Add transitions for deletion operations.) Deletion operations are represented by ε-transitions in a similar way as in the case of the PDA constructed by Algorithm 5.44. The only difference is that we do not use the pushdown store.

For each position $i \in \{2, \dots, |\boldsymbol{p}| - 2\}$ and number of errors $l \in \{0, \dots, k-1\}$: If $\boldsymbol{p}_i \in \Sigma$ and $l + \mathit{size} \leq k$, where $\mathit{size} = (\boldsymbol{S}_i^{\boldsymbol{p}} - i)/2$, then the proper bottom-up subtree of $\boldsymbol{p}$ that starts at position i can be deleted. This can be done by setting

$$\delta\left((i-1)^l, \varepsilon\right) \leftarrow \left\{(\boldsymbol{S}_i^{\boldsymbol{p}} - 1)^{l+\mathit{size}}\right\}.$$

9. (Return.) Return $\mathcal{M} = (Q, \Sigma_\uparrow, \delta, q_0, F)$.

▶ **Lemma 5.49** (Correctness of Algorithm 5.48)**.** *Given an ordered labeled tree $\mathcal{P}$ and $k \geq 0$, the ε-NFA constructed by Algorithm 5.48 accepts the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d_s)$.*

Proof. The lemma can be proved analogously as Lemma 5.45. However, since Algorithm 5.48 is very similar to Algorithm 5.44, we prove its correctness by showing that the two automata constructed by the above-mentioned algorithms accept the same language. The similarity between the two automata is as follows:

- Non-auxiliary states created in Step 2 of Algorithm 5.48 correspond to states created in Step 2 of Algorithm 5.44.
- The start state and the set of final states in both automata are the same since Steps 3 and 4 are the same in both algorithms.

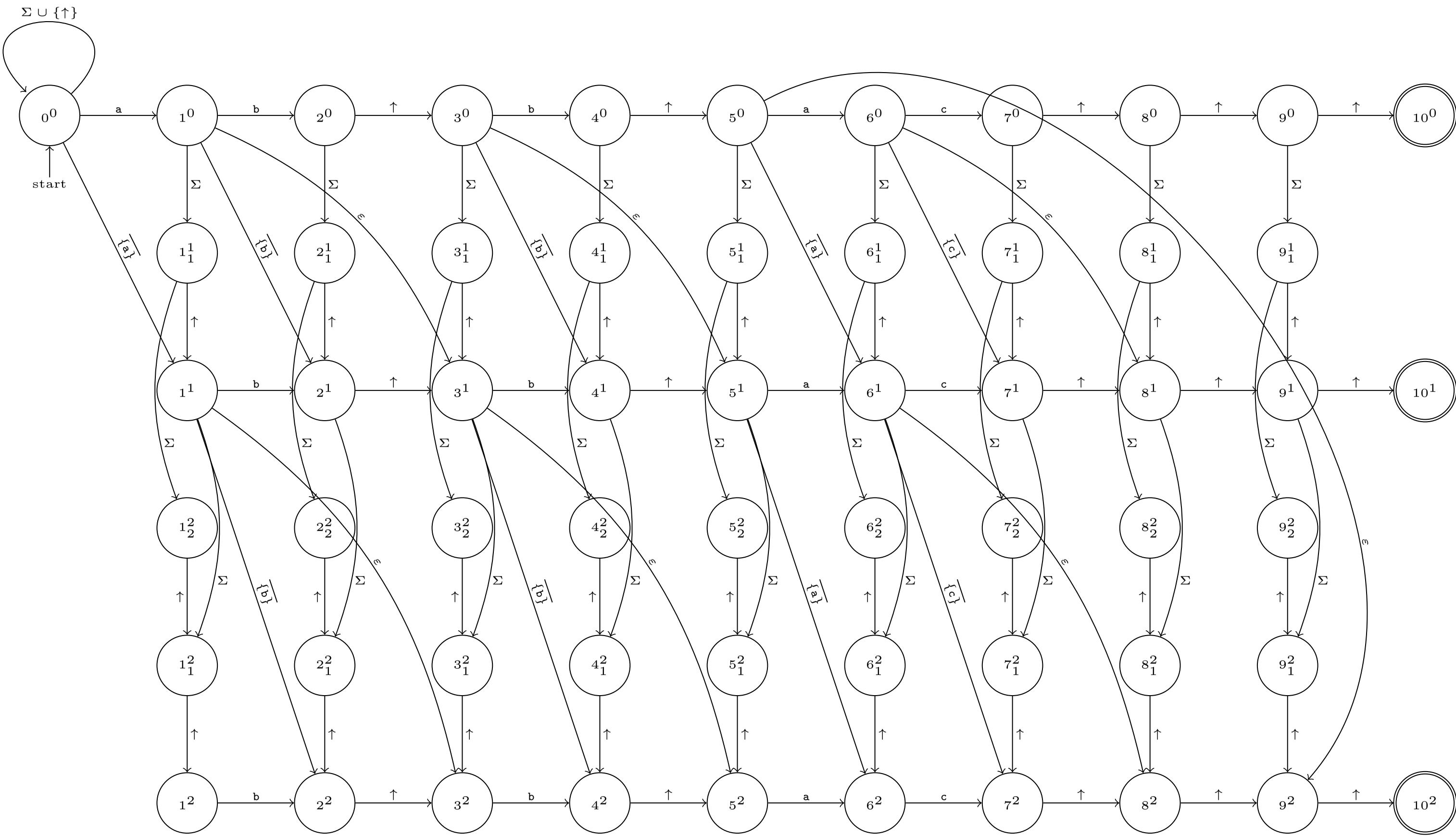


Figure 5.10 The ε-NFA constructed by Algorithm 5.48 accepting the pattern dictionary or the 1-degree matching ε-NFA for distance d_{s}, maximum number of errors $k = 2$, and pattern tree $\mathcal{P}$ illustrated Figure 5.1a. If the automaton contains loop transitions in the start state, it is the 1-degree matching automaton $\mathcal{M}(\mathsf{prefBar}(\mathcal{P}), k, d_{\mathrm{s}})$. If the loop transitions in the start state are removed, it is the ε-NFA accepting the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d_{\mathrm{s}})$.

- Transitions created in Step 5 of Algorithm 5.48 correspond to transitions created in Step 6 of Algorithm 5.44. The only difference is that the former transitions are not equipped with pushdown operations. However, the transitions created in Step 6 of Algorithm 5.44 do not change the content of the pushdown store. Thus, these transitions are the same. Similarly, transitions created in Step 6 of Algorithm 5.48 correspond to transitions created in Step 7 of Algorithm 5.44, and transitions created in Step 8 of Algorithm 5.48 correspond to transitions created in Step 9 of Algorithm 5.44.

Therefore, the algorithms differ only in two aspects: Step 2 of Algorithm 5.48 creates extra states called auxiliary, and the two algorithms create the transitions that correspond to insertion operations differently. However, we can observe that auxiliary states in the ε-NFA are used only by transitions corresponding to insertion operations. Therefore, our goal is to show that transitions created in Step 7 of Algorithm 5.48 correspond to transitions created in Step 8 of Algorithm 5.44.

In Algorithm 5.44, Step 8a corresponds to reading a node label (inserting a node): the level is increased by one, and the depth is not changed. Therefore, its corresponding bar must be read before the insertion operation is complete. This corresponds to pushing the symbol s to the pushdown store. Thus, the number of symbols s in the pushdown store corresponds to the number of bar symbols that must be read before the insertion operation is complete. Transitions that correspond to reading the bar symbol are created in Step 8b. Each of these transitions pops one symbol s from the pushdown store. These transitions do not change the level (or depth).

In Algorithm 5.48, Steps 7(a)i and 7(b)i correspond to Step 8a of Algorithm 5.44. Step 7(a)i creates transitions that read a node label corresponding to the root of the subtree that is being inserted. Transitions that read labels of other nodes are created in Step 7(b)i. Pushing symbol s to the pushdown store is in the ε-NFA simulated by the increased value s (the lower index) in the label of the target state (it is assumed that the non-auxiliary states have the lower index equal to zero). Similarly, the popping symbol s from the pushdown is simulated by decreasing the value s (the lower index) in the label of the target state. Steps 7(a)ii and 7(b)ii correspond to Step 8b of Algorithm 5.44. Step 7(a)ii creates transitions that read the bar symbol corresponding to the root of the subtree being inserted, and Step 7(b)ii creates transitions that read the remaining bar symbols.

Therefore, it is now easy to see that the transitions created in Step 7 of Algorithm 5.48 correspond to transitions created in Step 8 of Algorithm 5.44. Thus, the automaton created by Algorithm 5.48 accepts the same language, the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d_{\mathrm{s}})$, as the automaton created by Algorithm 5.44. ◀

Analogously as in the case of the 1-degree matching ε-NFA for $d_{\mathrm{s}}^{\mathrm{c}}$, there are two ways how to use the 1-degree matching ε-NFA for d_{s} as a basis for a matching algorithm. We can either transform it into an equivalent deterministic automaton or simulate it in a deterministic way. The former approach comes with a high space complexity. Since the number of states of the (ε-)NFA is $\Theta\big(k^2 \cdot |\boldsymbol{p}|\big)$, the trivial upper bound on the number of states is $\mathcal{O}\big(2^{k^2 \cdot |\boldsymbol{p}|}\big)$. We have not yet studied the non-trivial upper bound on the number of states of the equivalent deterministic automaton. An analysis analogous to the one described in Section 5.4.1 would be more complex, especially for the insertion operation. This is because we can insert an ordered labeled *forest* (a sequence of ordered labeled trees) of at most k nodes at each position inside the pattern string, and there are $\frac{1}{k+1}\binom{2k}{k}$ ordered (unlabeled) forests with k nodes [159, Theorem 6.2]. For labeled ordered forests, we need to consider that each of the forests comes in $|\Sigma|^k$ variants since there are k nodes, and each can have $|\Sigma|$ labels. Moreover, insertion can occur at several positions, but the total number of nodes inserted cannot exceed the maximum number of errors. For example, we can insert a forest of 3 nodes at position two and a forest of 2 nodes at position four if the maximum number of errors is greater than or equal to 5. An alternative approach to constructing a deterministic automaton is to simulate the ε-NFA in a deterministic way. In the following

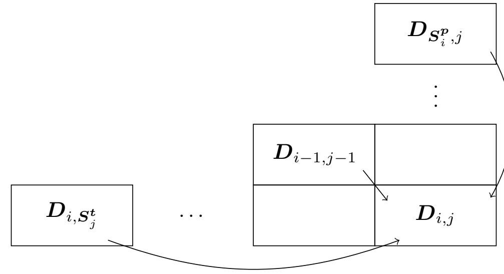


Figure 5.11 The value $\boldsymbol{D}_{i,j}$, where $i, j \geq 1$, depends on three positions at most: $\boldsymbol{D}_{i-1,j-1}$, $\boldsymbol{D}_{\boldsymbol{S}_i^{\boldsymbol{p}},j}$, and $\boldsymbol{D}_{i,\boldsymbol{S}_j^{\boldsymbol{t}}}$.

section, we propose a simulation based on dynamic programming. This algorithm can be seen as an extension of the algorithm presented for the distance d_s^c in Section 5.4.2.

5.5.3 Simulation of the 1-degree matching ε-NFA by dynamic programming

Similarly, as in the case of the constrained simple 1-degree edit distance, we can simulate the 1-degree matching ε-NFA for d_s in a deterministic way using a dynamic programming approach. The algorithm introduced in this section is an extension of the algorithm presented in Section 5.4.2. The extension lies in how insertion and deletion operations are handled since a bottom-up subtree of any size can now be inserted or deleted. We use the subtree jump table for both the pattern and the input tree to fill in the values in the array. This suggests that we need to read the input tree in the prefix bar notation twice: to build the subtree jump table and to compute the array used in dynamic programming. However, as we discuss later in this section, the subtree jump table can be computed on the fly while filling the array. Again, the algorithm reports end positions of pattern occurrences in the input, but start positions can also be (pre)computed. We describe the dynamic programming approach in Algorithm 5.50.

▶ **Algorithm 5.50** (Dynamic programming approach to the NFFOBI tree pattern matching problem under d_s)**.** Let Σ be an alphabet. Let $\mathcal{P}$ and $\mathcal{T}$ be ordered labeled trees over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$ and $\boldsymbol{t} = \mathsf{prefBar}(\mathcal{T})$. Let $k \geq 0$. Given $\boldsymbol{p}, \boldsymbol{t}$, and k, the algorithm finds every substring $\boldsymbol{t}_{j'\ldots j}$ of $\boldsymbol{t}$, where $1 \leq j' < j \leq |\boldsymbol{t}|$, such that $\boldsymbol{t}_{j'\ldots j} \in L(\Sigma, \uparrow)$ and $d_s(\boldsymbol{p}, \boldsymbol{t}_{j'\ldots j}) \leq k$. In other words, the algorithm finds every bottom-up subtree $\mathcal{S}$ in $\mathcal{T}$ such that $d_s(\mathcal{P}, \mathcal{S}) \leq k$.

The principal internal data structure is a two-dimensional array $\boldsymbol{D}$ in which the dimensions have ranges 0 to $|\boldsymbol{p}|$ and 0 to $|\boldsymbol{t}|$. By $\boldsymbol{D}_{i,j}$, where $i \in \{0, \ldots |\boldsymbol{p}|\}$ and $j \in \{0, \ldots, |\boldsymbol{t}|\}$, we denote the value at *row* i and *column* j. The value $\boldsymbol{D}_{i,j}$ with $i, j \geq 1$, depends on three positions at most; see Figure 5.11. When the array is filled, $\boldsymbol{D}_{|\boldsymbol{p}|,j}$ contains $d_s(\boldsymbol{p}, \boldsymbol{t}_{j'\ldots j})$, where $j' \in \{1, \ldots, j-1\}$, if and only if $\boldsymbol{t}_{j'\ldots j}$ represents the prefix bar notation of a bottom-up subtree of $\mathcal{T}$.

1. (Initialization.)
   a. Set $\boldsymbol{D}_{0,0} \leftarrow 0$.
   b. For each position i in $\boldsymbol{p}$, set $\boldsymbol{D}_{i,0} \leftarrow \infty$.
   c. For each position j in $\boldsymbol{t}$, set $\boldsymbol{D}_{0,j} \leftarrow 0$.
2. (Compute the subtree jump table for $\boldsymbol{p}$ and $\boldsymbol{t}$.)
   a. Compute the subtree jump table $\boldsymbol{S}^{\boldsymbol{p}}$ for $\boldsymbol{p}$ using Algorithm 5.42.
   b. Compute the subtree jump table $\boldsymbol{S}^{\boldsymbol{t}}$ for $\boldsymbol{t}$ using Algorithm 5.42.

3. (Fill array) For each position j in $\boldsymbol{t}$ and i in $\boldsymbol{p}$:

a. The cost of relabeling $\boldsymbol{p}_i$ to $\boldsymbol{t}_j$ (or match) is

$$c_1 \leftarrow \begin{cases} \boldsymbol{D}_{i-1,j-1} & \text{if } \boldsymbol{p}_i = \boldsymbol{t}_j, \\ \boldsymbol{D}_{i-1,j-1} + 1 & \text{if } \boldsymbol{p}_i \neq \boldsymbol{t}_j \wedge \boldsymbol{p}_i, \boldsymbol{t}_j \neq \uparrow, \\ \infty & \text{otherwise.} \end{cases}$$

b. The cost of deleting a substring $\boldsymbol{p}_{i'\dots i}$ from $\boldsymbol{p}$, where $i' \in \{2, \dots, i-1\}$ and $\boldsymbol{p}_{i'\dots i}$ represents a proper bottom-up subtree of $\mathcal{P}$, is

$$c_2 \leftarrow \begin{cases} \boldsymbol{D}_{pos,j} + size & \text{if } 3 \leq i < |\boldsymbol{p}| \wedge \boldsymbol{p}_i = \uparrow, \\ \infty & \text{otherwise,} \end{cases}$$

where

- $size = (i - \boldsymbol{S}^{\boldsymbol{p}}_i)/2$ represents the size of the bottom-up subtree $\boldsymbol{p}_{i'\dots i}$ and
- $pos = \boldsymbol{S}^{\boldsymbol{p}}_i$ represents the position of the symbol in $\boldsymbol{p}$ that immediately precedes the substring $\boldsymbol{p}_{i'\dots i}$; that is, the position $i' - 1$.

c. The cost of inserting a substring $\boldsymbol{t}_{j'\dots j}$ into $\boldsymbol{p}$ (at position $i+1$), where $2 \leq j' < j$ and $\boldsymbol{t}_{j'\dots j}$ represents a proper bottom-up subtree of $\mathcal{T}$, is

$$c_3 \leftarrow \begin{cases} \boldsymbol{D}_{i,pos} + size & \text{if } 3 \leq j < |\boldsymbol{t}| \wedge i < |\boldsymbol{p}| \wedge \boldsymbol{t}_j = \uparrow, \\ \infty & \text{otherwise,} \end{cases}$$

where

- $size = (j - \boldsymbol{S}^{\boldsymbol{t}}_j)/2$ represents the size of the bottom-up subtree $\boldsymbol{t}_{j'\dots j}$ and
- $pos = \boldsymbol{S}^{\boldsymbol{t}}_j$ represents the position of the symbol in $\boldsymbol{t}$ that immediately precedes the substring $\boldsymbol{t}_{j'\dots j}$; that is, the position $j' - 1$.

d. Set $\boldsymbol{D}_{i,j} \leftarrow \mathsf{min}(c_1, c_2, c_3)$. (We assume that $\infty > c$ and $\infty + c = \infty$ for every $c \in \mathbb{N}_0$ and that $\mathsf{min}(\infty, \infty, \infty) = \infty$.)

4. (Report occurrences.) For each position j in $\boldsymbol{t}$: If $\boldsymbol{D}_{|\boldsymbol{p}|,j} \leq k$, output j.

We show an example of our dynamic programming approach to the NFFOBI tree pattern matching problem in Figure 5.12. Similarly, as for the approach described in the previous section, if the last row contains a number, then the corresponding position in the input string always contains the bar symbol. However, this time the other implication is also true. If the corresponding position in the input string contains the bar symbol, then the last row contains a number. This is because an edit script always exists between the pattern tree and each bottom-up subtree of the input tree. The corresponding edit script can be computed similarly as described in the previous section.

We prove the correctness of our matching algorithm by showing that it is a simulation of the 1-degree matching ε-NFA for d_{s}.

▶ **Theorem 5.51** (Correctness of Algorithm 5.50)**.** *The dynamic programming approach described by Algorithm 5.50 is a simulation of the 1-degree matching ε-NFA for d_{s} that is constructed by adding a loop transition for every symbol $a \in \Sigma_{\uparrow}$ to the start state of the ε-NFA created by Algorithm 5.48.*

Proof. Let Σ be an alphabet. Let $\mathcal{P}$ and $\mathcal{T}$ be ordered labeled trees over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$ and $\boldsymbol{t} = \mathsf{prefBar}(\mathcal{T})$. Let $k \geq 0$. We recall that in the 1-degree matching ε-NFA for $\boldsymbol{p}$, k, and d_{s}, each state is labeled by i^l or i^l_s, where $i \in \{0, \dots, |\boldsymbol{p}|\}$, $l \in \{0, \dots, k\}$, and $s \in \{1, \dots, k\}$. We

$\boldsymbol{D}$	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
i	-	-	a	a	a	c	$\uparrow$	$\uparrow$	$\uparrow$	a	b	$\uparrow$	b	a	c	$\uparrow$	$\uparrow$	$\uparrow$	a	c	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	a	∞	0	0	0	1	1	2	3	0	1	1	1	0	1	1	3	4	0	1	1	6	10	∞
2	b	∞	∞	1	1	1	2	3	∞	4	0	5	1	2	1	3	3	8	5	1	6	10	∞	∞
3	$\uparrow$	∞	1	1	1	2	1	2	3	1	2	0	2	1	2	1	3	3	1	2	1	5	10	∞
4	b	∞	∞	2	2	2	3	4	∞	4	1	5	0	3	2	4	2	8	4	2	5	10	∞	∞
5	$\uparrow$	∞	2	2	2	3	2	3	4	2	3	1	3	2	3	2	4	2	2	3	2	4	10	∞
6	a	∞	∞	2	2	3	3	4	∞	4	3	5	2	3	3	4	4	8	2	3	3	10	∞	∞
7	c	∞	∞	∞	3	2	4	∞	∞	∞	5	∞	6	3	3	4	8	∞	9	2	10	∞	∞	∞
8	$\uparrow$	∞	∞	3	3	4	2	4	∞	5	4	5	3	4	4	3	4	8	3	4	2	10	∞	∞
9	$\uparrow$	∞	4	4	4	5	4	2	4	4	5	3	5	4	5	4	3	4	4	5	4	2	10	∞
10	$\uparrow$	∞	∞	∞	∞	∞	5	4	2	∞	∞	5	∞	∞	∞	5	4	3	∞	∞	5	4	2	10

Figure 5.12 An example of dynamic programming approach to the NFFOBI tree pattern matching problem under d_{s} that finds all occurrences of pattern tree $\mathcal{P}$ illustrated in Figure 5.1a in input tree $\mathcal{T}$ illustrated in Figure 5.1b with up to k errors. For $k = 2$, there are two occurrences of $\mathcal{P}$ in $\mathcal{T}$; the first found at position 7 of $\boldsymbol{t} = \mathsf{prefBar}(T)$ and the second found at position 21 of $\boldsymbol{t}$. The corresponding edit script can be obtained by tracking back the array from $\boldsymbol{D}_{10,7}$ to $\boldsymbol{D}_{0,1}$ and from $\boldsymbol{D}_{10,21}$ to $\boldsymbol{D}_{0,7}$. The substring $\boldsymbol{t}_{2\ldots7}$ corresponds to the bottom-up subtree $\mathcal{T}/\mathtt{a}^2$, and the edit script between $\mathcal{P}$ and $\mathcal{T}/\mathtt{a}^2$ consists of two deletion operations that delete the leaves from $\mathcal{P}$ labeled by b. The substring $\boldsymbol{t}_{8\ldots21}$ corresponds to the bottom-up subtree $\mathcal{T}/\mathtt{a}^5$, and the edit script between $\mathcal{P}$ and $\mathcal{T}/\mathtt{a}^5$ consists of two insertion operations that insert a bottom-up subtree of size 2, subtree ac $\uparrow\uparrow$, into $\mathcal{P}$.

say that a state labeled by i^l or i_s^l is at depth i and at level l; states labeled by i_s^l are called auxiliary. Specifically, states are labeled so they can be arranged into rows and columns similarly as illustrated in Figure 5.3. The difference is that there are more auxiliary states this time. In the case of $d_{\mathrm{s}}^{\mathrm{c}}$, there was one auxiliary row between each two consecutive main rows. This time, there is $l + 1$ auxiliary rows between main rows at level l and $l + 1$, see Figure 5.9. We also recall that the dynamic programming algorithm constructs a two-dimensional array $\boldsymbol{D}$ in which the dimensions have ranges 0 to $|\boldsymbol{p}|$ and 0 to $|\boldsymbol{t}|$, respectively. By $\boldsymbol{D}_{i,j}$, where $i \in \{0, \ldots |\boldsymbol{p}|\}$ and $j \in \{0, \ldots, |\boldsymbol{t}|\}$, we denote the value at row i and column j.

We prove that every value $\boldsymbol{D}_{i,j} \leq k$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j of the run of the ε-NFA. For $\boldsymbol{D}_{i,j}$ that contains ∞ or a number greater than k, we show that no non-auxiliary state is active in depth i of the ε-NFA and step j of the run of the ε-NFA. From this proof, it follows that $\boldsymbol{D}_{|\boldsymbol{p}|,j} \leq k$ if and only if the final state $|\boldsymbol{p}|^{\boldsymbol{D}_{|\boldsymbol{p}|,j}}$ is active in step j of the run of the ε-NFA and there is no $l < \boldsymbol{D}_{|\boldsymbol{p}|,j}$ such that state $|\boldsymbol{p}|^l$ is also active. In other words, the dynamic programming reports an occurrence with distance $l \in \{0, \ldots, k\}$ ending at position j in $\boldsymbol{t}$ if and only if the ε-NFA reports an occurrence with distance l ending at position j in $\boldsymbol{t}$.

We prove the claim mentioned above by a loop invariant proof using the following invariant: Before the start of iteration j' of the outer loop, for each $i \in \{0, \ldots, |\boldsymbol{p}|\}$ and $j'' \in \{0, \ldots, j' - 1\}$, value $\boldsymbol{D}_{i,j''}$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA is active.

Before the iteration of the first pass of the outer loop, $\boldsymbol{D}_{i,0}$ is initialized to 0 for $i = 0$ (Step 1a) and to ∞ for each $i \in \{1, \ldots, |\boldsymbol{p}|\}$ (Step 1b). At step 0 of the run of the ε-NFA, only state 0^0 is active. Thus, the claim holds for $j' = 1$.

If the loop invariant is true before the start of iteration j' of the outer loop, we show that it is true before the iteration $j'+1$. Before the iteration j', we get that for each $i \in \{0, \ldots, |\boldsymbol{p}|\}$ and $j'' \in \{0, \ldots, j'-1\}$, value $\boldsymbol{D}_{i,j''}$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA is active.

During iteration j' the inner loop is executed. We prove that for the inner loop, the following invariant holds: Before the start of iteration i' of the inner loop, for each $i'' \in \{0, \ldots, i'-1\}$, value $\boldsymbol{D}_{i'',j'}$ corresponds to the level of the topmost active non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA is active.

Before the iteration of the first pass of the inner loop, $\boldsymbol{D}_{0,j'}$ should contain the level of the topmost active non-auxiliary state in depth 0 of the ε-NFA and step j' of the run of the ε-NFA. At each step $j' \in \{0, \ldots, |\boldsymbol{t}|\}$ of the run of the ε-NFA, state 0^0 is active due to its loop transitions. Thus, the level of the topmost active non-auxiliary state in depth 0 of the ε-NFA and step j' of the run of the ε-NFA is 0. This corresponds to setting value $\boldsymbol{D}_{0,j'}$ to 0 in Step 1c for each $j' \in \{1, \ldots, |\boldsymbol{t}|\}$ and setting value $\boldsymbol{D}_{0,0}$ to 0 in Step 1a.

If the invariant is true before the start of iteration i' of the inner loop, we show that it is true before the iteration $i'+1$. Before the start of iteration i' of the inner loop, we get that for each $i'' \in \{0, \ldots, i'-1\}$, value $\boldsymbol{D}_{i'',j'}$ corresponds to the level of the topmost active non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA is active.

During iteration i', Step 3d is executed, and $\boldsymbol{D}_{i',j'}$ is set to $\mathsf{min}(c_1, c_2, c_3)$. We need to show that if there exists an active non-auxiliary state in depth i' of the ε-NFA and step j' of the run of the ε-NFA, then the level of the topmost one is equal to c_1, c_2, or c_3; otherwise, if there is no active non-auxiliary state in depth i' of the ε-NFA and step j' of the run of the ε-NFA, then c_1, c_2, and c_3 are equal to ∞ or contain value greater than k.

There are at most four types of transitions that lead to each non-auxiliary state in depth $i' \in \{1, \ldots, |\boldsymbol{p}|\}$ of the ε-NFA:

- a match transition (created in Step 5 of Algorithm 5.48),
- a relabeling transition (created in Step 6 of Algorithm 5.48),
- a subtree insertion "transition" (created in Step 7 of Algorithm 5.48; we can see a sequence of consecutive transitions leading through auxiliary states from one non-auxiliary state to another as one transition labeled by a string instead of a symbol), and
- a deletion transition (created in Step 8 of Algorithm 5.48).

Therefore, if there is an active non-auxiliary state in depth i' of the ε-NFA and step j' of the run of the ε-NFA, we get to it using the above mentioned transitions. However, not all transitions are always possible:

- The match transitions leading to depth i' can be used in step j' of the run of the ε-NFA only if there is an active non-auxiliary state in depth $i'-1$ and step $j'-1$ of the run of the ε-NFA and $\boldsymbol{t}_{j'} = \boldsymbol{p}_{i'}$.
- The relabeling transitions leading to depth i' can be used in step j' of the run of the ε-NFA only if there is an active non-auxiliary state in depth $i'-1$ and step $j'-1$ of the run of the ε-NFA and $\boldsymbol{t}_{j'} \neq \boldsymbol{p}_{i'}$ and $\boldsymbol{p}_{i'}, \boldsymbol{t}_{j'} \neq \uparrow$.
- The deletion transitions leading to depth i' for $i' \geq 3$ can be used in step j' of the run of the ε-NFA only if $\boldsymbol{p}_{i'} = \uparrow$ and there is an active non-auxiliary state in depth pos and step j' of the run of the ε-NFA, where $pos = \boldsymbol{S}^{\boldsymbol{p}}_{i'}$ represents the position of the symbol in $\boldsymbol{p}$ that immediately precedes the substring $\boldsymbol{p}_{\hat{\imath} \ldots i'}$ representing the corresponding bottom-up subtree in $\mathcal{P}$.

- The subtree insertion "transition" (a sequence of transitions created in Step 7 of Algorithm 5.48 leading to depth $i' \in \{1, \ldots, |\boldsymbol{p}| - 1\}$ can be used in step j' for $j' \geq 3$ of the run of the ε-NFA only if $\boldsymbol{t}_{j'} = \uparrow$ and there is an active non-auxiliary state in depth i' and step *pos* of the run of the ε-NFA, where $pos = \boldsymbol{S}^{\boldsymbol{t}}_{j'}$ represents the position of the symbol in $\boldsymbol{t}$ that immediately precedes the substring $\boldsymbol{t}_{\hat{j}\ldots j'}$ representing the corresponding bottom-up subtree in $\mathcal{T}$.

Since we are interested only in the topmost active non-auxiliary state in depth i' and step j' of the run of the ε-NFA, we can focus not only on the available transitions, but also only on those whose source is the topmost active state in the corresponding depth and step of the run of the ε-NFA. Moreover, if those transitions yield different states, we choose the one with the lowest level. During iteration i', the algorithm simulates exactly the steps mentioned above:

- Step 3a corresponds to checking whether matching or relabeling transition is possible (note that they are exclusive). If $\boldsymbol{p}_{i'} = \boldsymbol{t}_{j'}$, then c_1 is set to $\boldsymbol{D}_{i'-1,j'-1}$. Value $\boldsymbol{D}_{i'-1,j'-1}$ contains the level of the topmost active non-auxiliary state in depth $i'-1$ of the ε-NFA and step $j'-1$ of the run of the ε-NFA. In other words, in depth $i'-1$ of the ε-NFA and step $j'-1$ of the run of the ε-NFA the topmost active non-auxiliary state is $(i'-1)^l$, where $l = \boldsymbol{D}_{i'-1,j'-1}$. Since $\boldsymbol{p}_{i'} = \boldsymbol{t}_{j'}$, we can move to state $(i')^l$ using the match transition. If $\boldsymbol{p}_{i'} \neq \boldsymbol{t}_{j'}$ and $\boldsymbol{p}_{i'}, \boldsymbol{t}_{j'} \neq \uparrow$, then c_1 is set to $\boldsymbol{D}_{i'-1,j'-1} + 1$. This corresponds to moving from state $(i'-1)^l$ to state $(i')^{l+1}$ using the relabeling transition. If $c_1 > k$, then no state in depth i' and step j' of the run of the ε-NFA can be reached using a match or relabeling transition since there are only k levels. If $\boldsymbol{p}_{i'} \neq \boldsymbol{t}_{j'}$ and $\boldsymbol{p}_{i'}$ or $\boldsymbol{t}_{j'}$ is equal to the bar symbol, then no state in depth i' and step j' of the run of the ε-NFA can be reached using a match or relabeling transition. This corresponds to setting c_1 to ∞ in the dynamic programming algorithm.

- Step 3b corresponds to checking whether deletion transition is possible. If $3 \leq i' < |\boldsymbol{p}|$ and $\boldsymbol{p}_{i'} = \uparrow$, then c_2 is set to $\boldsymbol{D}_{pos,j'} + size$, where *size* represents the size of the bottom-up subtree of $\mathcal{P}$ that ends at position i' (the subtree that is being deleted). Value $\boldsymbol{D}_{pos,j'}$ contains the level of the topmost active non-auxiliary state in depth *pos* of the ε-NFA and step j' of the run of the ε-NFA, where $pos = \boldsymbol{S}^{\boldsymbol{p}}_{i'}$ represents the position of the symbol in $\boldsymbol{p}$ that immediately precedes the substring $\boldsymbol{p}_{\hat{i}\ldots i'}$ representing the bottom-up subtree that is being deleted. In other words, in depth *pos* of the ε-NFA and step j' of the run of the ε-NFA the topmost active non-auxiliary state is $(pos)^l$, where $l = \boldsymbol{D}_{pos,j'}$. Since $3 \leq i' < |\boldsymbol{p}|$ and $\boldsymbol{p}_{i'} = \uparrow$, we can move to state $(i')^{l+size}$ using the deletion transition. If $i' = |\boldsymbol{p}|$ or $\boldsymbol{p}_{i'} \neq \uparrow$, then no state in depth i' and step j' of the run of the ε-NFA can be reached using a deletion transition. This corresponds to setting c_2 to ∞ in the dynamic programming algorithm.

- Step 3c corresponds to checking whether a subtree insertion "transition" is possible. If $3 \leq j' < |\boldsymbol{t}|$, $i' < |\boldsymbol{p}|$, and $\boldsymbol{t}_{j'} = \uparrow$, then c_3 is set to $\boldsymbol{D}_{i',pos} + size$, where *size* represents the size of the bottom-up subtree of $\mathcal{T}$ that ends at position j' (the subtree that is being inserted into $\boldsymbol{p}$). Value $\boldsymbol{D}_{i',pos}$ contains the level of the topmost active non-auxiliary state in depth i' of the ε-NFA and step *pos* of the run of the ε-NFA, where $pos = \boldsymbol{S}^{\boldsymbol{t}}_{j'}$ represents the position of the symbol in $\boldsymbol{t}$ that immediately precedes the substring $\boldsymbol{t}_{\hat{j}\ldots j'}$ representing the bottom-up subtree that is being inserted. In other words, in depth i' of the ε-NFA and step *pos* of the run of the ε-NFA the topmost active non-auxiliary state is $(i')^l$, where $l = \boldsymbol{D}_{i',pos}$. Since $3 \leq j' < |\boldsymbol{t}|$, $i' < |\boldsymbol{p}|$, and $\boldsymbol{t}_{j'} = \uparrow$, we can move to state $(i')^{l+size}$ using the subtree insertion "transition" that leads through auxiliary states. If $j' < 3$ or $j' = |\boldsymbol{t}|$ or $i' = |\boldsymbol{p}|$, or $\boldsymbol{t}_{j'} \neq \uparrow$, then no state in depth i' and step j' of the run of the ε-NFA can be reached using a subtree insertion "transition". This corresponds to setting c_3 to ∞ in the dynamic programming algorithm.

- Step 3d executes $\boldsymbol{D}_{i',j'} \leftarrow \mathsf{min}(c_1, c_2, c_3)$. This clearly corresponds saving the level of the topmost active state in depth i' and step j' of the run of the ε-NFA (or saving value greater than k or ∞ if there is no such state).

Thus, before the start of iteration $i'+1$, $\boldsymbol{D}_{i'',j'}$ contains the level of the topmost active non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA for each $i'' \in \{0,\ldots,i'\}$. At termination $i = |\boldsymbol{p}|+1$, value $\boldsymbol{D}_{i'',j'}$ corresponds to the level of the topmost active non-auxiliary state in depth i'' of the ε-NFA and step j' of the run of the ε-NFA for each $i'' \in \{0,\ldots,|\boldsymbol{p}|\}$.

Thus, after the execution of the inner loop, we get that for each $i \in \{0,\ldots,|\boldsymbol{p}|\}$ and $j'' \in \{0,\ldots,j'\}$, value $\boldsymbol{D}_{i,j''}$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA or ∞ (or value greater than k) if no non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA is active. At termination $j = |\boldsymbol{t}|+1$, value $\boldsymbol{D}_{i,j''}$ corresponds to the level of the topmost active non-auxiliary state in depth i of the ε-NFA and step j'' of the run of the ε-NFA for each $i \in \{0,\ldots,|\boldsymbol{p}|\}$ and $j'' \in \{0,\ldots,|\boldsymbol{t}|\}$. Hence the dynamic programming simulates the 1-degree matching ε-NFA. ◀

Given an input tree and its prefix bar notation $\boldsymbol{t}$, Algorithm 5.50 computes the subtree jump table for $\boldsymbol{t}$ beforehand in Step 2b. However, as we show in Lemma 5.52, the subtree jump table can be computed on the fly while filling the array.

▶ **Lemma 5.52.** *Given the prefix bar notation $\boldsymbol{t}$ of an input tree, the values provided by the subtree jump table for $\boldsymbol{t}$ can be computed in Algorithm 5.50 on the fly so that Step 2b can be deleted.*

Proof. We modify Algorithm 5.50 and discuss its correctness. To avoid computing the subtree jump table for $\boldsymbol{t}$ beforehand, we use a stack $\boldsymbol{Y}$ with operations push that inserts a symbol on its top and pop that returns and deletes the top symbol, and we modify Algorithm 5.50 as follows:

1. (Initialization.)
   a. Set $\boldsymbol{D}_{0,0} \leftarrow 0$.
   b. For each position i in $\boldsymbol{p}$, set $\boldsymbol{D}_{i,0} \leftarrow \infty$.
   c. For each position j in $\boldsymbol{t}$, set $\boldsymbol{D}_{0,j} \leftarrow 0$.
   d. Create an empty stack $\boldsymbol{Y}$.
2. (Compute the subtree jump table for $\boldsymbol{p}$.) Compute the subtree jump table $\boldsymbol{S}^{\boldsymbol{p}}$ for $\boldsymbol{p}$ using Algorithm 5.42.
3. (Fill array) For each position j in $\boldsymbol{t}$:
   a. If $\boldsymbol{t}_j \neq \uparrow$, then push$(j)$ into $\boldsymbol{Y}$. Otherwise, $\mathit{value} \leftarrow$ pop$(\boldsymbol{Y})$.
   b. For each position i in $\boldsymbol{p}$:
      i. The cost of relabeling $\boldsymbol{p}_i$ to $\boldsymbol{t}_j$ (or match) is

$$c_1 \leftarrow \begin{cases} \boldsymbol{D}_{i-1,j-1} & \text{if } \boldsymbol{p}_i = \boldsymbol{t}_j, \\ \boldsymbol{D}_{i-1,j-1}+1 & \text{if } \boldsymbol{p}_i \neq \boldsymbol{t}_j \wedge \boldsymbol{p}_i, \boldsymbol{t}_j \neq \uparrow, \\ \infty & \text{otherwise.} \end{cases}$$

      ii. The cost of deleting a substring $\boldsymbol{p}_{i'\ldots i}$ from $\boldsymbol{p}$, where $i' \in \{2,\ldots,i-1\}$ and $\boldsymbol{p}_{i'\ldots i}$ represents a proper bottom-up subtree of $\mathcal{P}$, is

$$c_2 \leftarrow \begin{cases} \boldsymbol{D}_{\mathit{pos},j} + \mathit{size} & \text{if } 3 \leq i < |\boldsymbol{p}| \wedge \boldsymbol{p}_i = \uparrow, \\ \infty & \text{otherwise,} \end{cases}$$

      where
      - $\mathit{size} = (i - \boldsymbol{S}^{\boldsymbol{p}}_i)/2$ represents the size of the bottom-up subtree $\boldsymbol{p}_{i'\ldots i}$ and
      - $\mathit{pos} = \boldsymbol{S}^{\boldsymbol{p}}_i$ represents the position of the symbol in $\boldsymbol{p}$ that immediately precedes the substring $\boldsymbol{p}_{i'\ldots i}$; that is, the position $i'-1$.

iii. The cost of inserting a substring $\boldsymbol{t}_{j'\dots j}$ into $\boldsymbol{p}$ (at position $i+1$), where $2 \le j' < j$ and $\boldsymbol{t}_{j'\dots j}$ represents a proper bottom-up subtree of $\mathcal{T}$, is

$$c_3 \leftarrow \begin{cases} \boldsymbol{D}_{i,pos} + size & \text{if } 3 \le j < |\boldsymbol{t}| \wedge i < |\boldsymbol{p}| \wedge \boldsymbol{t}_j = \uparrow, \\ \infty & \text{otherwise,} \end{cases}$$

where
- $pos = value - 1$ represents the position of the symbol in $\boldsymbol{t}$ that immediately precedes the substring $\boldsymbol{t}_{j'\dots j}$; that is, the position $j'-1$, and
- $size = (j - pos)/2$ represents the size of the bottom-up subtree $\boldsymbol{t}_{j'\dots j}$.

iv. Set $\boldsymbol{D}_{i,j} \leftarrow \mathsf{min}(c_1, c_2, c_3)$. (We assume that $\infty > c$ and $\infty + c = \infty$ for every $c \in \mathbb{N}_0$ and that $\mathsf{min}(\infty,\infty,\infty) = \infty$.)

4. (Report occurrences.) For each position j in $\boldsymbol{t}$: If $\boldsymbol{D}_{|\boldsymbol{p}|,j} \le k$, output j.

The main difference between Algorithm 5.50 and its modified version mentioned above is the computation of the cost for insertion operation; compare Step 3(b)iii above and Step 3c in Algorithm 5.50. In Algorithm 5.50, the variables *pos* and *size* for insertion operations are computed from $\boldsymbol{S}^{\boldsymbol{t}}_j$. In the modified algorithm, the same values are obtained from stack $\boldsymbol{Y}$. The equality of these values follows from Algorithm 5.42. ◀

Additionally, we can apply the same improvements to Algorithm 5.50 as we did to Algorithm 5.38 in the previous section:

- All values $\boldsymbol{D}_{i,j} > k$ can be replaced by the value ∞ representing a number of errors greater than k. This is because such values can no longer produce an occurrence.
- Step 4 of Algorithm 5.50 can be merged with Step 3 so that we report occurrences as we fill the array instead of waiting until the whole array is filled.

We now discuss the time and space complexity of Algorithm 5.50.

▶ **Theorem 5.53** (Time complexity of Algorithm 5.50)**.** *Let $\mathcal{P}$ and $\mathcal{T}$ be ordered labeled trees with m and n nodes, respectively. Given the prefix bar notation of $\mathcal{P}$ and $\mathcal{T}$ and $k \ge 0$, the time complexity of Algorithm 5.50 is $\mathcal{O}(mn)$.*

Proof. The algorithm builds the subtree jump table for $\mathsf{prefBar}(\mathcal{P})$ and $\mathsf{prefBar}(\mathcal{T})$ in time $\Theta(m)$ and $\Theta(n)$, respectively. Then, a two-dimensional array $\boldsymbol{D}$ of size $(2m+1)\cdot(2n+1)$ is built. The computation of the value at each position of $\boldsymbol{D}$ depends on three positions at most and this computation executes in constant time. ◀

▶ **Theorem 5.54** (Space complexity of Algorithm 5.50)**.** *Let $\mathcal{P}$ and $\mathcal{T}$ be ordered labeled trees with m and n nodes, respectively. Assume $m \le n$. Given the prefix bar notation of $\mathcal{P}$ and $\mathcal{T}$ and $k \ge 0$, Algorithm 5.50 can be implemented to use $\mathcal{O}(km)$ space.*

Proof. The dynamic programming approach builds a two-dimensional array $\boldsymbol{D}$ of size $(2m+1)\cdot(2n+1)$. The computation of the value at each position of $\boldsymbol{D}$ depends on three positions at most: $\boldsymbol{D}_{i-1,j-1}$, $\boldsymbol{D}_{\boldsymbol{S}^{\boldsymbol{p}}_i,j}$, and $\boldsymbol{D}_{i,\boldsymbol{S}^{\boldsymbol{t}}_j}$. Since algorithm performs the computation column-wise, values in $\boldsymbol{S}^{\boldsymbol{p}}$ are accessed repeatedly. Thus, the algorithm precomputes these values (Step 2) and stores them in the array of size $2m$. To compute column $j \in \{1,\dots,|\boldsymbol{t}|\}$ (values $\boldsymbol{D}_{i,j}$ for $i \in \{0,\dots,|\boldsymbol{p}|\}$), we need values in column given by $\boldsymbol{S}^{\boldsymbol{t}}_j$. However, storing the whole subtree jump table for $\boldsymbol{t}$ is not necessary as we described in the proof of Lemma 5.52. The necessary values can be computed on the fly using stack $\boldsymbol{Y}$. Still, $\boldsymbol{Y}$ needs n space in the worst case (the case where $\mathcal{T}$ is a linear tree). However, $\boldsymbol{Y}$ is used only to compute the start position of possibly inserted subtree. Thus, the length (number of stored elements) in $\boldsymbol{Y}$ can be limited to k (implemented by

removing the oldest element from the bottom of the stack). In other words, we can restrict the number of previous columns that are needed to compute column j by $2k$ because values in the remaining preceding columns correspond to a situation that a tree of size larger than k is being inserted into $\boldsymbol{p}$ and these insertions can be ignored. Therefore, $O(km)$ space is needed in order to compute all values. ◀

We note that we also proposed an alternative version of Algorithm 5.50 [157], [158]. The main idea behind the alternative algorithm is to simulate insertion operations transition-by-transition. This is done by adding one dimension to the array, so we use a three-dimensional array instead of the two-dimensional one. The third dimension range is from 0 to k, and values 1 to k are used to simulate insertion transitions that lead to auxiliary states. As a result, the subtree jump table for the input tree is not needed, and the algorithm can be implemented to use $\mathcal{O}(km)$ space. However, the time complexity of this approach is $\mathcal{O}(kmn)$, where k is the maximum number of errors allowed, m is the number of nodes in the pattern tree, and n is the number of nodes in the input tree.

We recall that at the end of Section 5.4.2, we discussed that the dynamic programming algorithm cannot be modified so that the match operation is outside the minimum function. The same also holds for Algorithm 5.50. For example, in Figure 5.12, it is more favorable to apply deletion operation (that deletes the subtree `ac` ↑↑ from the pattern) when computing value $\boldsymbol{D}_{9,10}$ than to copy value $\boldsymbol{D}_{8,9}$ which corresponds to match.

5.6 1-degree matching automaton for the (constrained) 1-degree edit distance

In this section, we pay attention to adapting our methods proposed in the previous two sections to the variant where different costs may accompany the edit operations. First, we focus on the constrained 1-degree edit distance and then on the 1-degree edit distance. Given two ordered labeled trees over alphabet Σ, we assume that the edit operations are assigned their costs using a metric cost function on Σ_λ. Moreover, we assume that costs are nonnegative integers. For examples presented in this section, we assume that 1-degree edit operations are assigned their costs using the metric cost function described by Table 5.1.

The main idea behind the adaptation of constructing the 1-degree matching automaton resides in changing the target states for transitions corresponding to the edit operations. For example, in matching under d_s^c and d_s, applying a relabeling operation always corresponds to using a transition from a state at level $l \in \{0, \dots, k-1\}$ to a state at level $l+1$, where k is the maximum number of errors allowed. When operations are assigned non-unit costs, transitions must lead into states at a level increased accordingly by the cost of the operation.

The main idea behind adapting the dynamic programming algorithm lies in changing the value that is added when an edit operation can be applied to the current position. In matching under d_s^c and d_s, we always add one. In the case of non-unit cost operations, we add the operation cost instead.

In the following section, we discuss an adaptation of our methods proposed in Section 5.4 to the case where operations are assigned non-unit costs. Then, in Section 5.6.2, we follow a similar approach for the 1-degree edit distance.

5.6.1 Constrained 1-degree edit distance

In this section, we focus on the NFFOBI tree pattern matching problem under the constrained 1-degree edit distance; see Problem 5.14. Our solution is an extension of algorithms presented in Section 5.4, in which we focused on the simpler variant of this problem where no costs were

involved. First, we describe a modification of the 1-degree matching automaton. Then, we describe a modification of the dynamic programming algorithm.

5.6.1.1 1-degree matching ε-NFA for the constrained 1-degree edit distance

Let Σ be alphabet. Let γ be a metric cost function defined on Σ_λ. To construct the 1-degree matching ε-NFA for d^c, we make three changes to Algorithm 5.28:

- We change Step 5 (Add transitions for relabeling operations.) as follows:
  - For each $a \in \Sigma \setminus \{\boldsymbol{p}_1\}$: If $\gamma(\boldsymbol{p}_1, a) \leq k$, then set $\delta\big(0^0, a\big) \leftarrow \big\{1^{\gamma(\boldsymbol{p}_1,a)}\big\}$.
  - For each position $i \in \{2, \ldots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \ldots, k-1\}$: If $\boldsymbol{p}_i \neq \uparrow$, then for each $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$ such that $l + \gamma(\boldsymbol{p}_i, a) \leq k$, set $\delta\big((i-1)^l, a\big) \leftarrow \big\{i^{l+\gamma(\boldsymbol{p}_i,a)}\big\}$.
- We change Step 6 (Add transitions for insertion operations.) as follows:
  - For each position $i \in \{2, \ldots, |\boldsymbol{p}|\}$, number of errors $l \in \{0, \ldots, k-1\}$, and symbol $a \in \Sigma$ such that $l + \gamma(\lambda, a) \leq k$:
    1. Set $\delta\big((i-1)^l, a\big) \leftarrow \delta\big((i-1)^l, a\big) \cup \big\{(i-1)_1^{l+\gamma(\lambda,a)}\big\}$.
    2. Set $\delta\big((i-1)_1^{l+\gamma(\lambda,a)}, \uparrow\big) \leftarrow \big\{(i-1)^{l+\gamma(\lambda,a)}\big\}$.
- We change Step 7 (Add transitions for deletion operations.) as follows:
  - For each position $i \in \{2, \ldots, |\boldsymbol{p}|-2\}$ and number of errors $l \in \{0, \ldots, k-1\}$: If $\boldsymbol{p}_i \in \Sigma$, $\boldsymbol{p}_{i+1} = \uparrow$ and $l + \gamma(\boldsymbol{p}_i, \lambda) \leq k$, then set $\delta\big((i-1)^l, \varepsilon\big) \leftarrow \big\{(i+1)^{l+\gamma(\boldsymbol{p}_{i-1},\lambda)}\big\}$.

By applying the changes described above to Algorithm 5.28, we construct an ε-NFA accepting the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d^c)$ for a pattern tree $\mathcal{P}$ and a given maximum number of errors k. To transform this automaton into the 1-degree matching ε-NFA for d^c, we add a loop transition to its start state for every possible input symbol. These modifications do not change the number of states of the automaton nor the construction time complexity. Therefore, the number of states of the 1-degree matching ε-NFA is $|\boldsymbol{p}| + 1 + k(2|\boldsymbol{p}| - 1)$ and the automaton can be built in time $\mathcal{O}(k \cdot |\boldsymbol{p}|)$. However, we note that some states can be unreachable and thus, can be removed. For example, if $\gamma(\lambda, a) \geq 2$ for each $a \in \Sigma$, then no incoming transition leads to auxiliary states i_1^1, where $i \in \{1, \ldots, |\boldsymbol{p}| - 1\}$. Moreover, if the cost of each operation is greater than or equal to some value $l \leq k$, then all states at levels 1 to $l - 1$ are unreachable.

The proof of the correctness of algorithms that build the ε-NFA accepting the pattern dictionary for d^c and the 1-degree matching ε-NFA for d^c is similar to the proof of Lemma 5.29 and Lemma 5.31, respectively. For brevity, we do not include them. We show an example of the 1-degree matching ε-NFA for d^c in Figure 5.13.

5.6.1.2 Simulation of the 1-degree matching ε-NFA for the constrained 1-degree edit distance by dynamic programming

The 1-degree matching ε-NFA for d^c can be used again as a basis for the matching algorithm in two ways: by transforming it into an equivalent deterministic automaton or by simulating it in a deterministic way. In this dissertation thesis, we do not study the size of the corresponding DFA. We only say that its size is not greater than the size of the 1-degree matching DFA for d_s^c. This is because there cannot be a greater number of strings created using individual edit operations. For example, in the case of 1-degree matching DFA for d_s^c, we can relabel at most k symbols and each to $|\Sigma|$ options. In the case of 1-degree matching DFA for d^c there are no more variants. On the contrary, using k relabeling operations might be impossible since these operations can have a cost greater than or equal to 2. In this section, we discuss how the dynamic programming algorithm can be modified for the case where operations have non-unit costs.

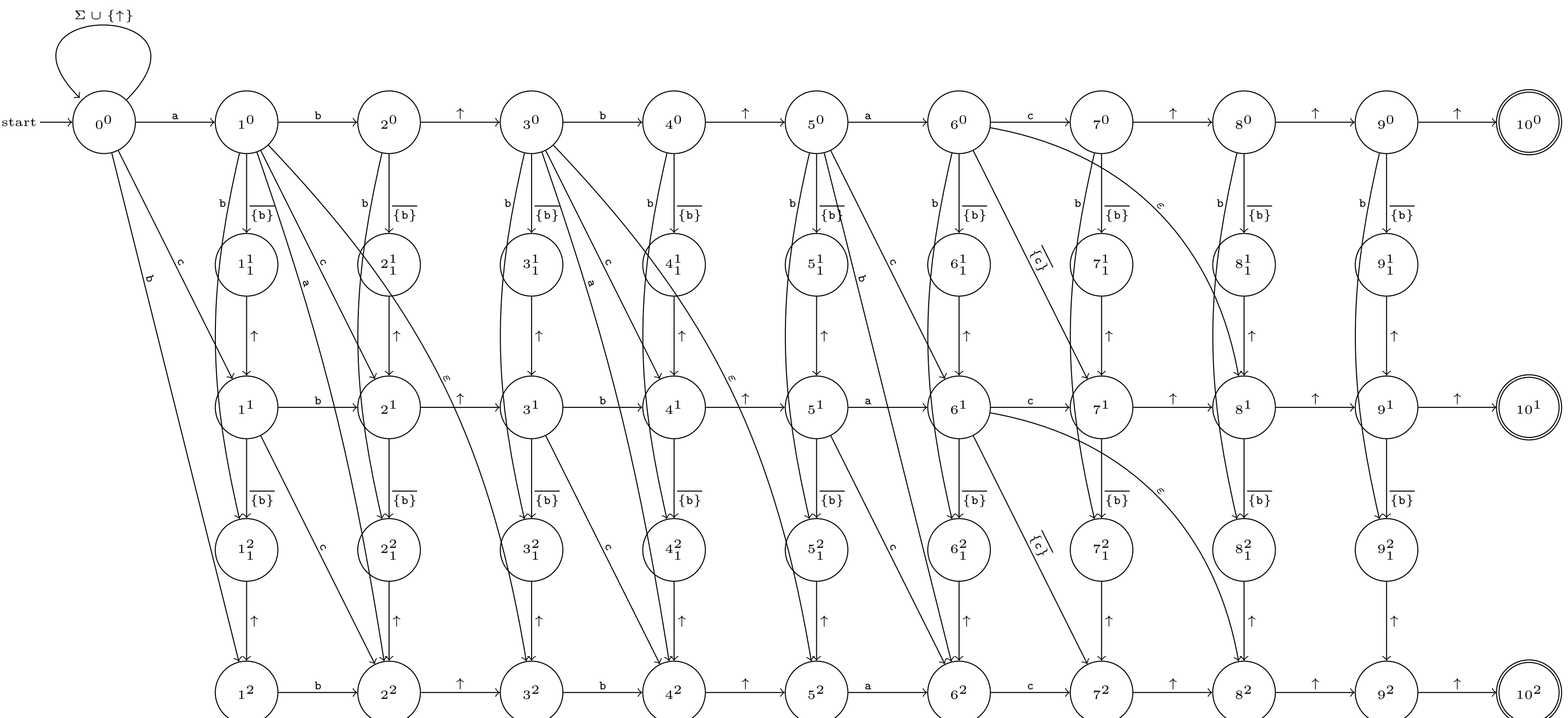


Figure 5.13 The 1-degree matching ε-NFA for distance d^c, maximum number of errors $k = 2$, and pattern tree $\mathcal{P}$ illustrated Figure 5.1a. The 1-degree edit operations are assigned their costs using the metric cost function described in Table 5.1.

Let Σ be alphabet. Let γ be a metric cost function defined on Σ_λ. A simulation of the 1-degree matching ε-NFA for d^c can be based on the dynamic programming approach described by Algorithm 5.38 by changing computation of c_1, c_2, and c_3 as follows:

Step 2a The cost of relabeling $\boldsymbol{p}_i$ to $\boldsymbol{t}_j$ (or match) is

$$c_1 \leftarrow \begin{cases} \boldsymbol{D}_{i-1,j-1} & \text{if } \boldsymbol{p}_i = \boldsymbol{t}_j, \\ \boldsymbol{D}_{i-1,j-1} + \gamma(\boldsymbol{p}_i, \boldsymbol{t}_j) & \text{if } \boldsymbol{p}_i \neq \boldsymbol{t}_j \wedge \boldsymbol{p}_i, \boldsymbol{t}_j \neq \uparrow, \\ \infty & \text{otherwise.} \end{cases}$$

Step 2b The cost of deleting substring $\boldsymbol{p}_{i-1}\boldsymbol{p}_i$ from $\boldsymbol{p}$ is

$$c_2 \leftarrow \begin{cases} \boldsymbol{D}_{i-2,j} + \gamma(\boldsymbol{p}_{i-1}, \lambda) & \text{if } i \geq 3 \wedge \boldsymbol{p}_i = \uparrow \wedge \boldsymbol{p}_{i-1} \neq \uparrow, \\ \infty & \text{otherwise.} \end{cases}$$

Step 2c The cost of inserting substring $\boldsymbol{t}_{j-1}\boldsymbol{t}_j$ into $\boldsymbol{p}$ (at position $i+1$) is

$$c_3 \leftarrow \begin{cases} \boldsymbol{D}_{i,j-2} + \gamma(\lambda, \boldsymbol{t}_{j-1}) & \text{if } j \geq 3 \wedge \boldsymbol{t}_j = \uparrow \wedge \boldsymbol{t}_{j-1} \neq \uparrow \wedge i < |\boldsymbol{p}|, \\ \infty & \text{otherwise.} \end{cases}$$

By applying these changes to Algorithm 5.38, we obtain a dynamic programming algorithm for the NFFOBI tree pattern matching problem under d^c. Its correctness can be proved similarly as the correctness of Algorithm 5.38. Moreover, we can apply the same improvements as we did for Algorithm 5.38:

- All values $\boldsymbol{D}_{i,j} > k$ in the array can be replaced by the value ∞ representing a number of errors greater than k.
- Occurrences can be reported on the fly while filling the array instead of waiting until the whole array is filled.
- Since every column can be computed using just two previous columns, the algorithm can be implemented to use only $\mathcal{O}(m)$ space, where m is the number of nodes of a given pattern tree.

It is also easy to see that the time and space complexity of the extended dynamic programming approach described in this section remains the same as those given by Theorem 5.40 and Theorem 5.41.

5.6.2 1-degree edit distance

The NFFOBI tree pattern matching problem under the 1-degree edit distance is the most advanced of the four problems defined in Section 5.2. Let us recall that the 1-degree edit distance between two ordered labeled trees is given by the cost of a least-cost edit script between them. We present a solution to this problem as an extension of the solution proposed in Section 5.5. First, we present an adaptation of the 1-degree matching PDA. Then, we show an adaption of the 1-degree matching ε-NFA. Finally, we discuss how the dynamic programming approach can be modified to simulate the 1-degree matching ε-NFA for distance d.

5.6.2.1 1-degree matching PDA for the 1-degree edit distance

The construction of the 1-degree matching PDA for distance d can be based on a modification of Algorithm 5.44. If operations have non-unit costs, each transition that corresponds to an edit operation must lead to a target state with a level increased accordingly by the cost of the

	a	b	↑	b	↑	a	c	↑	↑	↑
	1	2	3	4	5	6	7	8	9	10
1	11	4	1	6	3	10	9	6	5	0
2	1	3	3	5	5	6	7	7	7	7

Figure 5.14 The weighted subtree jump table for $\mathsf{prefBar}(\mathcal{P})$, where $\mathcal{P}$ is the tree illustrated in Figure 5.1a, and 1-degree edit operations are assigned costs according to the cost function γ described in Table 5.1.

operation. Such target states can be easily computed for relabeling and insertion operations since the corresponding transitions handle one node at a time. However, a deletion transition can represent multiple deletion operations since we are allowed to delete a proper bottom-up subtree of any size up to a given maximum number of errors. Therefore, to correctly determine target states for these transitions, we need to know the cost of deleting proper bottom-up subtrees of a given tree. For this purpose, we introduce a new data structure called a *weighted subtree jump table*, an extension of the subtree jump table; see Definition 5.55. The extension is based on adding another dimension to the table that can be used to compute the cost of deleting bottom-up subtrees. We illustrate an example of the weighted subtree jump table in Figure 5.14.

▶ **Definition 5.55** (Weighted subtree jump table)**.** Let Σ be an alphabet. Let $\boldsymbol{x} \in L(\Sigma, \uparrow)$. Let γ be a cost function defined on Σ_λ. Moreover, we define $\gamma(\uparrow, \lambda) = 0$. The *weighted subtree jump table* for $\boldsymbol{x}$ is a two-dimensional array $\boldsymbol{W}^{\boldsymbol{x}}$ in which the dimensions have ranges 1 to 2 and 1 to $|\boldsymbol{x}|$, respectively. Values in $\boldsymbol{W}^{\boldsymbol{x}}$ are defined as follows:

- $\boldsymbol{W}^{\boldsymbol{x}}_{1,i} = j + 1$ and $\boldsymbol{W}^{\boldsymbol{x}}_{1,j} = i - 1$ for each substring $\boldsymbol{x}_{i\ldots j} \in L(\Sigma, \uparrow)$, where $1 \leq i < j \leq |\boldsymbol{x}|$.
- $\boldsymbol{W}^{\boldsymbol{x}}_{2,i} = \sum_{j=1}^{i} \gamma(\boldsymbol{x}_j, \lambda)$ for each $i \in \{1, \ldots, |\boldsymbol{x}|\}$.

In Lemma 5.56, we explain how the weighted subtree jump table can be used to compute the cost of deleting individual proper bottom-up subtrees of a given tree. Note that if the cost function satisfies the symmetry condition, then for each bottom-up subtree, the cost of its deletion is the same as the cost of its insertion.

▶ **Lemma 5.56** (Computation of the cost of deleting a proper bottom-up subtree using the weighted subtree jump table)**.** *Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $\boldsymbol{p} =$* $\mathsf{prefBar}(\mathcal{P})$*. Let γ be a cost function defined on Σ_λ. Let $\boldsymbol{W}^{\boldsymbol{p}}$ be the weighted subtree jump table for $\boldsymbol{p}$. Let $\boldsymbol{p}_{i'\ldots i}$, where $1 < i' < i < |\boldsymbol{p}|$ represents a proper bottom-up subtree of $\mathcal{P}$. Then, the cost of deleting this subtree can be computed as follows:*

$$\boldsymbol{W}^{\boldsymbol{p}}_{2,i} - \boldsymbol{W}^{\boldsymbol{p}}_{2,i'-1}.$$

Proof. Recall that $\boldsymbol{W}^{\boldsymbol{p}}_{2,i} = \sum_{j=1}^{i} \gamma(\boldsymbol{p}_j, \lambda)$. It holds that

$$\boldsymbol{W}^{\boldsymbol{p}}_{2,i} - \boldsymbol{W}^{\boldsymbol{p}}_{2,i'-1} = \sum_{j=1}^{i} \gamma(\boldsymbol{p}_j, \lambda) - \sum_{j=1}^{i'-1} \gamma(\boldsymbol{p}_j, \lambda) = \sum_{j=i'}^{i} \gamma(\boldsymbol{p}_j, \lambda).$$

Since we defined $\gamma(\uparrow, \lambda) = 0$, the last sum represents the total cost of deleting all symbols from $\boldsymbol{p}_{i'\ldots i}$ that correspond to node labels in the subtree $\boldsymbol{p}_{i'\ldots i}$. Therefore, the sum computes the cost of deleting this subtree. ◀

We now present an algorithm for computing the weighted subtree jump table for the prefix bar notation of a given ordered labeled tree. The algorithm is an extension of Algorithm 5.42. The only difference is that we additionally need to compute values in the second dimension. The time complexity of the extended algorithm remains the same as the time complexity of Algorithm 5.42; it constructs the weighted subtree jump table in time linear to the length of a given string that represents the prefix bar notation of an ordered labeled tree.

▶ **Algorithm 5.57** (Computation of the weighted subtree jump table for an ordered labeled tree in the prefix bar notation)**.** Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Let γ be a cost function defined on Σ_λ. Moreover, we define $\gamma(\uparrow, \lambda) = 0$. Given $\boldsymbol{p}$ and γ, the algorithm constructs the weighted subtree jump table for $\boldsymbol{p}$ which is represented as a two-dimensional array $\boldsymbol{W^p}$ with ranges 1 to 2 and 1 to $|\boldsymbol{p}|$, respectively. The principal internal data structure is a stack $\boldsymbol{Y}$ with operations push that inserts a symbol on its top and pop that returns and deletes the top symbol.

1. (Initialize.)
   a. Create an empty stack $\boldsymbol{Y}$.
   b. Create a two-dimensional array $\boldsymbol{W^p}$ with ranges 1 to 2 and 1 to $|\boldsymbol{p}|$, respectively.
   c. Set $sum = 0$.
2. (Fill array.) For each position i in $\boldsymbol{p}$:
   a. Set $sum \leftarrow sum + \gamma(\boldsymbol{p}_i, \lambda)$.
   b. Set $\boldsymbol{W}^{\boldsymbol{p}}_{2,i} \leftarrow sum$.
   c. If $\boldsymbol{p}_i \neq\uparrow$, then $\mathsf{push}(i)$ into $\boldsymbol{Y}$. Otherwise, set $\boldsymbol{W}^{\boldsymbol{p}}_{1,i} \leftarrow \mathsf{pop}(\boldsymbol{Y}) - 1$ and set $\boldsymbol{W}^{\boldsymbol{p}}_{1,i'} \leftarrow i + 1$, where $i' = \boldsymbol{W}^{\boldsymbol{p}}_{1,i} + 1$.
3. (Return.) Return $\boldsymbol{W^p}$.

▶ **Lemma 5.58** (Correctness of Algorithm 5.57)**.** *Let Σ be an alphabet. Let $\mathcal{P}$ be an ordered labeled tree over Σ. Let $\boldsymbol{p} = \mathsf{prefBar}(\mathcal{P})$. Let γ be a cost function defined on Σ_λ. Moreover, we define $\gamma(\uparrow, \lambda) = 0$. Given $\boldsymbol{p}$ and γ, Algorithm 5.57 constructs the weighted subtree jump table for $\boldsymbol{p}$.*

Proof. It is easy to see that $\boldsymbol{W}^{\boldsymbol{p}}_1$ is the same as $\boldsymbol{S^p}$ constructed by Algorithm 5.42. Thus, the correctness of $\boldsymbol{W}^{\boldsymbol{p}}_1$ follows from Lemma 5.43.

Recall that by definition $\boldsymbol{W}^{\boldsymbol{p}}_{2,i} = \sum_{j=1}^{i} \gamma(\boldsymbol{p}_j, \lambda)$ for each $i \in \{1, \dots, |\boldsymbol{p}|\}$. We prove correctness for $\boldsymbol{W}^{\boldsymbol{p}}_2$ by a loop invariant proof using the following invariant: Before the start of iteration l of the loop, the variable sum contains $\sum_{j=1}^{l-1} \gamma(\boldsymbol{p}_j, \lambda)$. The loop invariant is true before the first iteration of the loop (true for $i = 1$). Before the iteration of the first pass of the loop, sum is initialized to zero which is $\sum_{j=1}^{0} \gamma(\boldsymbol{p}_j, \lambda)$. Thus, the variable sum holds the correct sum before the first pass of the loop. If the loop invariant is true before the iteration l, we show that it is true before the iteration $l+1$. Before the iteration l of the loop, we get that sum contains $\sum_{j=1}^{l-1} \gamma(\boldsymbol{p}_j, \lambda)$. During iteration l, we execute Step 2a so that sum contains $\sum_{j=1}^{l-1} \gamma(\boldsymbol{p}_j, \lambda) + \gamma(\boldsymbol{p}_l, \lambda)$. Thus, before the start of iteration $l+1$, the variable sum holds $\sum_{j=1}^{l} \gamma(\boldsymbol{p}_j, \lambda)$. At termination, $i = |\boldsymbol{p}| + 1$, the variable sum holds $\sum_{j=1}^{|\boldsymbol{p}|} \gamma(\boldsymbol{p}_j, \lambda)$ which is the desired value for $\boldsymbol{W}^{\boldsymbol{p}}_{2,|\boldsymbol{p}|}$. Hence the algorithm correctly computes values in $\boldsymbol{W}^{\boldsymbol{p}}_2$. ◀

We are now in a position to introduce an algorithm that constructs the 1-degree matching PDA for d. Let Σ be alphabet. Let γ be a metric cost function defined on Σ_λ. To construct the 1-degree matching PDA for d, we make four changes to Algorithm 5.44:

- In Step 1, we compute the weighted subtree jump table $\boldsymbol{W^p}$ for $\boldsymbol{p}$ using Algorithm 5.57 instead of the subtree jump table $\boldsymbol{S^p}$ for $\boldsymbol{p}$.
- We change Step 7 (Add transitions for relabeling operations.) as follows:
  1. For each $a \in \Sigma \setminus \{\boldsymbol{p}_1\}$: If $\gamma(\boldsymbol{p}_1, a) \leq k$, then set $\delta\big(0^0, a, \bot\big) \leftarrow \big\{\big(1^{\gamma(\boldsymbol{p}_1, a)}, \bot\big)\big\}$.
  2. For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \dots, k-1\}$: If $\boldsymbol{p}_i \neq\uparrow$, then for each $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$ such that $l + \gamma(\boldsymbol{p}_i, a) \leq k$, set $\delta\big((i-1)^l, a, \bot\big) \leftarrow \big\{\big(i^{l+\gamma(\boldsymbol{p}_i, a)}, \bot\big)\big\}$.

- We change Step 8 (Add transitions for insertion operations.) as follows:
  - For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$, number of errors $l \in \{0, \dots, k-1\}$, and symbol $a \in \Sigma$ such that $l + \gamma(\lambda, a) \leq k$:
    1. Set $\delta\big((i-1)^l, a, \varepsilon\big) \leftarrow \big\{\big((i-1)^{l+\gamma(\lambda,a)}, \mathtt{s}\big)\big\}$.
    2. Set $\delta\big((i-1)^{l+\gamma(\lambda,a)}, \uparrow, \mathtt{s}\big) \leftarrow \big\{\big((i-1)^{l+\gamma(\lambda,a)}, \varepsilon\big)\big\}$.
- We change Step 9 (Add transitions for deletion operations.) as follows: Instead of setting the level of a target state using the size of the proper bottom-up subtree being deleted, we set the level using the cost of deleting the subtree. The cost is given by the weighted subtree jump table using Lemma 5.56.
  - For each position $i \in \{2, \dots, |\boldsymbol{p}| - 2\}$ and number of errors $l \in \{0, \dots, k-1\}$: If $\boldsymbol{p}_i \in \Sigma$ and $l + cost \leq k$, where
$$cost = \boldsymbol{W}^{\boldsymbol{p}}_{2, \boldsymbol{W}^{\boldsymbol{p}}_{1,i} - 1} - \boldsymbol{W}^{\boldsymbol{p}}_{2, i-1},$$
then the proper bottom-up subtree of $\boldsymbol{p}$ that starts at position i can be deleted. This can be done by setting
$$\delta\left((i-1)^l, \varepsilon, \bot\right) \leftarrow \left\{\left((\boldsymbol{W}^{\boldsymbol{p}}_{1,i} - 1)^{l+cost}, \bot\right)\right\}.$$

By applying the changes described above to Algorithm 5.44, we construct a pushdown automaton accepting the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d)$ for a pattern tree $\mathcal{P}$ and a given maximum number of errors k. By adding a loop transition to its start state for every possible input symbol, we transform it into the 1-degree matching PDA for d. For the same reasons as we discussed in Section 5.6.1, it can happen that some states of the pushdown automaton are unreachable. These states can be removed. However, the 1-degree matching PDA for d has the same number of states as the 1-degree matching PDA for d_{s} in the worst case; that is, $|\boldsymbol{p}| + 1 + k(2|\boldsymbol{p}| - 1)$ states. The automaton can be built in time $\mathcal{O}(k \cdot |\boldsymbol{p}|)$. The proof of the correctness of the algorithm that builds the pushdown automaton accepting the pattern dictionary for d and the algorithm that constructs the 1-degree matching PDA for d is analogous to the proof of Lemma 5.45 and Theorem 5.47, respectively. We do not include these proofs for brevity. In Figure 5.15, we show an example of the 1-degree matching PDA for d.

5.6.2.2 1-degree matching ε-NFA for the 1-degree edit distance

We now proceed to a construction of the 1-degree matching ε-NFA for d. The construction is analogous to the construction of the 1-degree matching ε-NFA for d_{s}—instead of using the pushdown store for the insertion operations, we use auxiliary states. This is possible since the size of the pushdown store is again bounded by a given maximum number of errors k. Moreover, if costs assigned to insertion operations are higher than 1, then the size of the pushdown store is bounded by a number smaller than k. Therefore, the 1-degree matching ε-NFA for d can in some cases need smaller number of auxiliary states than the number of states of the 1-degree matching ε-NFA for d_{s}. In the worst case, the number of states of the 1-degree matching ε-NFA for d is the same as the number of states of the 1-degree matching ε-NFA for d_{s}; that is, $1 + (k+1) \cdot |\boldsymbol{p}| + (|\boldsymbol{p}| - 1) \cdot \frac{k(k+1)}{2} = \Theta\big(k^2 \cdot |\boldsymbol{p}|\big)$ states.

Let Σ be alphabet. Let γ be a metric cost function defined on Σ_λ. To construct the 1-degree matching ε-NFA for d, we make four changes to Algorithm 5.48:

- In Step 1, we compute the weighted subtree jump table $\boldsymbol{W}^{\boldsymbol{p}}$ for $\boldsymbol{p}$ using Algorithm 5.57 instead of the subtree jump table $\boldsymbol{S}^{\boldsymbol{p}}$ for $\boldsymbol{p}$.
- We change Step 6 (Add transitions for relabeling operations.) as follows:
  1. For each $a \in \Sigma \setminus \{\boldsymbol{p}_1\}$: If $\gamma(\boldsymbol{p}_1, a) \leq k$, then set $\delta\big(0^0, a\big) \leftarrow \big\{1^{\gamma(\boldsymbol{p}_1, a)}\big\}$.

Figure 5.15 The 1-degree matching PDA for distance d, maximum number of errors $k = 2$, and pattern tree $\mathcal{P}$ illustrated Figure 5.1a. The 1-degree edit operations are assigned their costs using the metric cost function described in Table 5.1.

2. For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$ and number of errors $l \in \{0, \dots, k-1\}$: If $\boldsymbol{p}_i \neq \uparrow$, then for each $a \in \Sigma \setminus \{\boldsymbol{p}_i\}$ such that $l + \gamma(\boldsymbol{p}_i, a) \leq k$, set $\delta\big((i-1)^l, a\big) \leftarrow \big\{i^{l+\gamma(\boldsymbol{p}_i, a)}\big\}$.

- We change Step 7 (Add transitions for insertion operations.) as follows:

1. (Add transitions from/into states at main rows.) For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$, number of errors $l \in \{0, \dots, k-1\}$, and symbol $a \in \Sigma$ such that $l + \gamma(\lambda, a) \leq k$:

   a. Set $\delta\big((i-1)^l, a\big) \leftarrow \delta\big((i-1)^l, a\big) \cup \big\{(i-1)_1^{l+\gamma(\lambda,a)}\big\}$.

   b. Set $\delta\big((i-1)_1^{l+\gamma(\lambda,a)}, \uparrow\big) \leftarrow \big\{(i-1)^{l+\gamma(\lambda,a)}\big\}$.

2. (Add transitions between auxiliary states.) For each position $i \in \{2, \dots, |\boldsymbol{p}|\}$, number of errors $l \in \{1, \dots, k-1\}$, value $s \in \{1, \dots, l\}$, and symbol $a \in \Sigma$ such that $l + \gamma(\lambda, a) \leq k$:

   a. Set $\delta\big((i-1)_s^l, a\big) \leftarrow \big\{(i-1)_{s+1}^{l+\gamma(\lambda,a)}\big\}$.

   b. Set $\delta\big((i-1)_{s+1}^{l+\gamma(\lambda,a)}, \uparrow\big) \leftarrow \big\{(i-1)_s^{l+\gamma(\lambda,a)}\big\}$.

- We change Step 8 (Add transitions for deletion operations.) as follows:

   – For each position $i \in \{2, \dots, |\boldsymbol{p}|-2\}$ and number of errors $l \in \{0, \dots, k-1\}$: If $\boldsymbol{p}_i \in \Sigma$ and $l + cost \leq k$, where

$$cost = \boldsymbol{W}^{\boldsymbol{p}}_{2,\boldsymbol{W}^{\boldsymbol{p}}_{1,i}-1} - \boldsymbol{W}^{\boldsymbol{p}}_{2,i-1},$$

   then the proper bottom-up subtree of $\boldsymbol{p}$ that starts at position i can be deleted. This can be done by setting

$$\delta\big((i-1)^l, \varepsilon\big) \leftarrow \big\{(\boldsymbol{W}^{\boldsymbol{p}}_{1,i} - 1)^{l+cost}\big\}\,.$$

By applying the changes described above to Algorithm 5.48, we construct an ε-NFA accepting the pattern dictionary $L(\mathsf{prefBar}(\mathcal{P}), k, d)$ for a pattern tree $\mathcal{P}$, maximum number of errors k, and distance d. The automaton is built in time $\mathcal{O}\big(k^2 \cdot |\boldsymbol{p}|\big)$. The proof of the correctness of the algorithm can be based on reasoning similar to that presented in the proof of Lemma 5.49. By adding a loop transition for every possible input symbol to the start state of the ε-NFA, we create the 1-degree matching ε-NFA for d. See Figure 5.16 for an example.

5.6.2.3 Simulation of the 1-degree matching ε-NFA for the 1-degree edit distance by dynamic programming

We can use the 1-degree matching ε-NFA for d as a basis for the matching algorithm by transforming it into a DFA or simulating it in a deterministic way. In this dissertation thesis, we focus on the latter approach. We present an algorithm based on dynamic programming that can be seen as a simulation of the ε-NFA. Given a pattern tree $\mathcal{P}$ and an input tree $\mathcal{T}$ in the prefix bar notation, the algorithm uses the weighted subtree jump table for both $\mathsf{prefBar}(\mathcal{P})$ and $\mathsf{prefBar}(\mathcal{T})$. The former table gives us the cost of deleting each proper bottom-up subtree of $\mathcal{P}$, and the latter table gives the cost of inserting each proper bottom-up subtree of $\mathcal{T}$ into $\mathcal{P}$. Note that the weighted subtree jump table for $\mathsf{prefBar}(\mathcal{T})$ contains in fact the cost of subtree deletion not insertion. However, these costs are equal since we assume that a given cost function is a metric.

Let Σ be alphabet. Let γ be a metric cost function defined on Σ_λ. A simulation of the 1-degree matching ε-NFA for d can based on the dynamic programming approach described by Algorithm 5.50 by applying four changes:

- In Step 1, we compute the weighted subtree jump table $\boldsymbol{W}^{\boldsymbol{p}}$ for $\boldsymbol{p}$ and $\boldsymbol{W}^{\boldsymbol{t}}$ for $\boldsymbol{t}$ using Algorithm 5.57 instead of the subtree jump table $\boldsymbol{S}^{\boldsymbol{p}}$ for $\boldsymbol{p}$ and $\boldsymbol{S}^{\boldsymbol{t}}$ for $\boldsymbol{t}$.

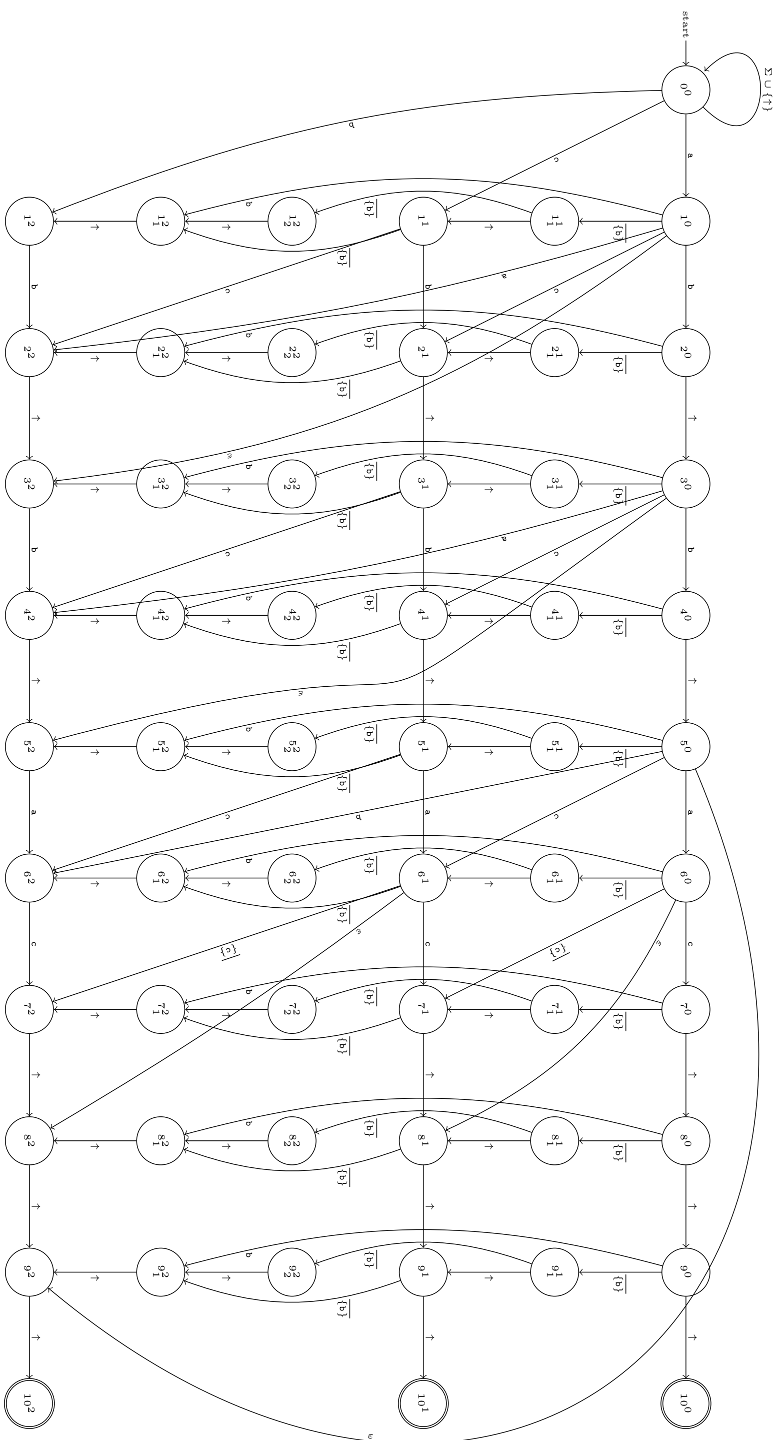


Figure 5.16 The 1-degree matching ε-NFA for distance d, maximum number of errors $k = 2$, and pattern tree $\mathcal{P}$ illustrated Figure 5.1a. The 1-degree edit operations are assigned their costs using the metric cost function described in Table 5.1.

- We change Step 3a, the computation of the cost of relabeling $\boldsymbol{p}_i$ to $\boldsymbol{t}_j$ (or match), as follows:

$$c_1 \leftarrow \begin{cases} \boldsymbol{D}_{i-1,j-1} & \text{if } \boldsymbol{p}_i = \boldsymbol{t}_j, \\ \boldsymbol{D}_{i-1,j-1} + \gamma(\boldsymbol{p}_i, \boldsymbol{t}_j) & \text{if } \boldsymbol{p}_i \neq \boldsymbol{t}_j \wedge \boldsymbol{p}_i, \boldsymbol{t}_j \neq \uparrow, \\ \infty & \text{otherwise.} \end{cases}$$

- We change Step 3b, the computation of the cost of deleting a substring $\boldsymbol{p}_{i'\dots i}$ from $\boldsymbol{p}$, where $i' \in \{2, \dots, i-1\}$ and $\boldsymbol{p}_{i'\dots i}$ represents a proper bottom-up subtree of $\mathcal{P}$, as follows:

$$c_2 \leftarrow \begin{cases} \boldsymbol{D}_{pos,j} + cost & \text{if } 3 \leq i < |\boldsymbol{p}| \wedge \boldsymbol{p}_i = \uparrow, \\ \infty & \text{otherwise,} \end{cases}$$

  where

  - $cost = \boldsymbol{W}^{\boldsymbol{p}}_{2,i} - \boldsymbol{W}_{2,\boldsymbol{W}^{\boldsymbol{p}}_{1,i}}$ represents the cost of deleting the bottom-up subtree $\boldsymbol{p}_{i'\dots i}$ and
  - $pos = \boldsymbol{W}^{\boldsymbol{p}}_{1,i}$ represents the position of the symbol in $\boldsymbol{p}$ that immediately precedes the substring $\boldsymbol{p}_{i'\dots i}$; that is the position $i' - 1$.

- We change Step 3c, the computation of the cost of inserting a substring $\boldsymbol{t}_{j'\dots j}$ into $\boldsymbol{p}$ (at position $i+1$), where $2 \leq j' < j$ and $\boldsymbol{t}_{j'\dots j}$ represents a proper bottom-up subtree of $\mathcal{T}$, as follows:

$$c_3 \leftarrow \begin{cases} \boldsymbol{D}_{i,pos} + cost & \text{if } 3 \leq j < |\boldsymbol{t}| \wedge i < |\boldsymbol{p}| \wedge \boldsymbol{t}_j = \uparrow, \\ \infty & \text{otherwise,} \end{cases}$$

  where

  - $cost = \boldsymbol{W}^{\boldsymbol{t}}_{2,j} - \boldsymbol{W}_{2,\boldsymbol{W}^{\boldsymbol{t}}_{1,j}}$ represents the cost of inserting the bottom-up subtree $\boldsymbol{t}_{j'\dots j}$ and
  - $pos = \boldsymbol{W}^{\boldsymbol{t}}_{1,j}$ represents the position of the symbol in $\boldsymbol{t}$ immediately preceding the substring $\boldsymbol{t}_{j'\dots j}$; that is the position $j' - 1$.

By applying these changes to Algorithm 5.50, we obtain a dynamic programming algorithm for the NFFOBI tree pattern matching problem under d. Its correctness can be proved analogously as the correctness of Algorithm 5.50. Moreover, we can apply the same improvements as we did for Algorithm 5.50:

- All values $\boldsymbol{D}_{i,j} > k$ in the array can be replaced by the value ∞ representing a number of errors greater than k.
- Occurrences can be reported on the fly while filling the array instead of waiting until the whole array is filled.
- The weighted subtree jump table for $\boldsymbol{t}$ can be computed on the fly so that string $\boldsymbol{t}$ is by the algorithm read only once. First, values in the first dimension correspond to values of the ordinary subtree jump table and these values can be computed on the fly; see Lemma 5.52. Second, each value in the second dimension depends only on preceding values.
- For similar reasons as discussed in the proof of Theorem 5.54, every column can be computed using at most $2k$ previous columns. Thus, we do not need to store the whole weighted subtree jump table for $\boldsymbol{t}$ but only the previous $2k$ values.

Using these improvements, it is easy to see that the algorithm can be implemented so that the time and space complexity of the dynamic programming approach remain the same as those given by Theorems 5.53 and 5.54, respectively.

We note that we also proposed an alternative version of the simulation of the 1-degree matching ε-NFA for d by dynamic programming [158]. The main idea behind the alternative algorithm is

the same as the idea of the alternative version of the simulation of the 1-degree matching ε-NFA for d_{s} that we described at the end of Section 5.5.3. The alternative approach does not need the weighted subtree jump table for the input tree. It can be implemented to use $\mathcal{O}(km)$ space. However, the time complexity of this approach is $\mathcal{O}(kmn)$, where k is the maximum number of errors allowed, m is the number of nodes in the pattern tree, and n is the number of nodes in the input tree.

5.7 Summary

Inspired by techniques for the problem of inexact string pattern matching described in Section 4.1.1, we showed that the string automata approach could also be used to solve the inexact tree pattern matching problem. Specifically, we focused on the online version of the NFFOBI tree pattern matching problem for ordered unranked trees. To measure the similarity between trees, we used four different edit distances: the constrained simple 1-degree edit distance $d_{\mathrm{s}}^{\mathrm{c}}$, the constrained 1-degree edit distance d^{c}, the simple 1-degree edit distance d_{s}, and the 1-degree edit distance d. The first two distances are novel variants of the 1-degree edit distance in which deletion and insertion operations cannot be used recursively to insert or delete a subtree of any size. We introduced the constrained simple 1-degree edit distance and the constrained 1-degree edit distance in Section 5.1.

We based our approach on reducing the NFFOBI tree pattern matching problem to a string matching problem; see Problem 5.59. For string representation of trees, we used the prefix bar notation. It follows from the substring property of this notation that if a pattern tree $\mathcal{P}$ matches a bottom-up subtree $\mathcal{S}$ of an input tree $\mathcal{T}$, then $\mathsf{prefBar}(\mathcal{P})$ matches $\mathsf{prefBar}(\mathcal{S})$, which is a substring of $\mathsf{prefBar}(\mathcal{T})$. To recognize that $\mathsf{prefBar}(\mathcal{P})$ matches $\mathsf{prefBar}(\mathcal{S})$, we defined four novel (string) edit distances so that the distance between two strings encoding trees is the same as the corresponding (tree) edit distance between the underlying trees.

▶ **Problem 5.59** (NFFOBI tree pattern matching problem for ordered unranked trees in the prefix bar notation)**.** Let Σ be an alphabet. Let $\boldsymbol{t}, \boldsymbol{p} \in L(\Sigma, \uparrow)$. Let $k \geq 0$. Let f be equal either to $d_{\mathrm{s}}, d_{\mathrm{s}}^{\mathrm{c}}, d$, or d^{c}. Given $\boldsymbol{t}, \boldsymbol{p}, k$, and f, the NFFOBI tree pattern matching problem under f for ordered unranked trees in the prefix bar notation is to find every substring $\boldsymbol{t}_{j' \ldots j}$ of $\boldsymbol{t}$, where $1 \leq j' < j \leq |\boldsymbol{t}|$, such that $\boldsymbol{t}_{j' \ldots j} \in L(\Sigma, \uparrow)$ and $f(\boldsymbol{p}, \boldsymbol{t}_{j' \ldots j}) \leq k$.

Our proposed solution extends arbology research by adapting the principles of the string automata approach to the inexact string pattern matching in trees. In the inexact string pattern matching, the goal is to find positions j in the text $\boldsymbol{t}$ such that there is a suffix of $\boldsymbol{t}_{1,\ldots,j}$ that matches a given pattern $\boldsymbol{p}$ with k or fewer errors. The same holds for our inexact tree pattern matching problem. However, it follows from the prefix bar notation that each of the reported positions corresponds to a position of the bar symbol in the input tree. This is because the position corresponds to the end position of a pattern occurrence, and each such occurrence is a bottom-up subtree of the input tree whose encoding always ends with the bar symbol. For similar reasons, each occurrence always starts at a position that does not contain the bar symbol. The start position for each occurrence can be easily computed in time linear to its size, or they can be precomputed beforehand while transforming the input tree to its prefix bar notation. Moreover, if the original input tree structure is available, we can connect each bar symbol in its prefix bar notation to the corresponding node. Thus, instead of reporting end positions in the input string as occurrences, we can return the root nodes of the corresponding bottom-up subtrees.

Given a pattern tree $\mathcal{P}$ and a maximum number of allowed errors k, the main idea of our automata-based approach was to identify the pattern dictionary, the set of all strings representing the trees whose distance from $\mathcal{P}$ is at most k. Then, we built a dictionary automaton called a 1-degree matching automaton.

Because the pattern dictionary is always a finite language, it is possible to construct the 1-degree matching automaton as a finite automaton. To find the positions of all the occurrences

of the pattern tree in a given input tree $\mathcal{T}$, the 1-degree matching automaton is run on $\mathsf{prefBar}(\mathcal{T})$. The automaton then reports a match every time it goes through a final state. In other words, the automaton can locate all occurrences of a pattern tree in any given input tree. However, we note that a finite automaton cannot recognize whether the input string is the valid prefix bar representation of a tree. This is because the language $L(\Sigma, \uparrow)$ is not regular.

For each of the four variants of the NFFOBI tree pattern matching problems, we presented an algorithm that constructed the 1-degree matching automaton as an ε-NFA. This automaton can then be transformed into an equivalent DFA. Assuming that the time necessary to compute a transition from each state is constant, the DFA can locate all pattern occurrences in time that is linear to the input size. However, this approach comes with high state complexity that limits its practicality. In Section 5.4.1, we proved a non-trivial upper bound on the state complexity of the 1-degree matching DFA for $d_{\mathrm{s}}^{\mathrm{c}}$, which is $\mathcal{O}\big(|\Sigma|^k \cdot k \cdot m^{k+1}\big)$, where m is the number of nodes of the pattern tree and k is the maximum number of errors allowed.

As an alternative approach, we presented an algorithm based on dynamic programming for each of the four variants of the NFFOBI tree pattern matching problem. As discussed in Section 4.1.2, dynamic programming is the most commonly used approach for computing tree edit distance and solving inexact tree pattern matching problems. However, existing methods lack clear references to a systematic approach to the standard theory of formal languages and automata. We showed that each of our dynamic programming algorithms could be seen as a way of simulating the corresponding 1-degree matching ε-NFA. Given a pattern tree and an input tree with m and n nodes, respectively, our dynamic programming approach comes with $\mathcal{O}(mn)$ time complexity and $\mathcal{O}(mk)$ space complexity. For the NFFOBI tree pattern matching problem under $d_{\mathrm{s}}^{\mathrm{c}}$, the space complexity is $\mathcal{O}(m)$.

In this chapter, we assumed trees to be unranked. We note that ranked trees can be also represented using the prefix bar notation and processed using the methods proposed in this chapter. However, insertion and deletion operations change the arity of the parent of the node that is being inserted or deleted. Thus, applying a set of edit operations can produce an invalid ranked tree. This issue can be addressed, for example, by changing the set of edit operations so that each insertion and deletion operations also change the parent's label. Moreover, relabeling operations should only change the label of a node to a label with the same rank. This can be a subject of further research.

In the domain of string automata approach to the problem of inexact tree pattern matching, we suggest the following directions for future research:

- an experimental evaluation of algorithms introduced in this chapter,
- an exploration of bit parallelism as a way of simulating the proposed nondeterministic finite automata,
- an investigation of the properties of the dynamic programming array and an exploration of whether the properties can be used to improve the algorithms introduced in this chapter,
- an exploration of automata approach to inexact tree pattern matching under other edit distances and for different types of trees, and
- further inspection of the space complexity of the deterministic finite automata.

Moreover, to our knowledge, there are no known solutions to the inexact tree pattern matching problems that are based on tree automata. Thus, future research can address this gap.

Chapter 6
Main results in tree indexing

In this chapter, we focus on the problem of tree pattern matching, where patterns are linear and allowed to contain path wildcards. Given such a pattern, we look for its exact occurrences in top-down subtrees or subtrees of a given input tree. Specifically, using the SNINWE classification terminology, we focus on three tree pattern matching problems: LFFOTE, LFFOSE, and L(PW)FOTE. Apart from these problems, we also propose a solution to a variant of the L(PW)FOTE tree pattern matching problem where pattern trees are assumed to be *fully-gapped*; that is, they contain the maximum number of path wildcards. Our primary focus is on ordered unranked trees. However, in the last section of this chapter, we briefly discuss adjusting our methods for trees that are ranked, unordered, or both.

We assume that all tree pattern matching problems considered in this chapter come in the offline variant. In other words, we focus on indexed searching: a given input tree is preprocessed and its index, which can be used to speed the search later, is built. Specifically, we propose four indexing structures that are based on string automata. Namely, we present

rootpath automaton that is an index of all the rootpaths of an ordered labeled tree,

path automaton that is an index of all the paths of an ordered labeled tree,

fully-gapped rootpath automaton that is an index of an ordered labeled tree for every linear fully-gapped tree for which there exists a matching rootpath in the indexed tree, and

gapped rootpath automaton that is an index of an ordered labeled tree for every linear gapped tree for which there exists a matching rootpath in the indexed tree.

Our proposed solutions extend arbology research by adapting the principles of the automata approach to string indexing discussed in Section 4.2.1 for trees.

The content of this chapter is partially based on our two conference papers [160], [161] and a journal article [162]. However, in those publications, we tailored the methods discussed in this chapter specifically to XML. In this chapter, we present our results more generally considering arbitrary ordered labeled (unranked) trees. The explanation of individual methods is also much more detailed, and we present them in the context of the SNINWE classification. Moreover, the path automaton is only described in this dissertation thesis and not in the above-mentioned publications.

Before we introduce our automata-based indexing methods, we first formally define the problems that they aim to solve; see Section 6.1. Then, in Section 6.2, we provide a general overview of our automata approach. In the four sections that follow, we present our automata-based indexes: rootpath automaton in Section 6.3, path automaton in Section 6.4, fully-gapped

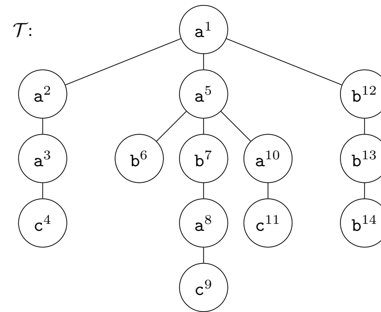


Figure 6.1 An ordered labed tree $\mathcal{T}$ over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ that we use throughout this chapter as an example of the input tree that is being indexed.

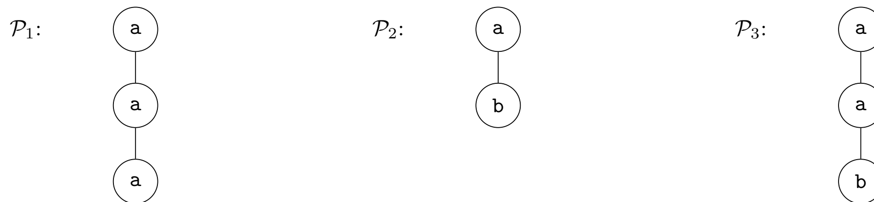


Figure 6.2 Linear ordered labeled (pattern) trees over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

rootpath automaton in Section 6.5, and gapped rootpath automaton in Section 6.6. Finally, we conclude this chapter with a summary in Section 6.7.

Throughout this chapter, we demonstrate our approach on the input tree illustrated in Figure 6.1.

6.1 Problem statement

This chapter focuses on the exact tree pattern matching problem for ordered labeled trees and linear patterns. In this section, we formally define four variants of this problem based on whether pattern trees are allowed to contain path wildcards and whether we look for occurrences in the top-down subtrees or subtrees of the input tree. We use the SNINWE classification introduced in Section 3.1 to define these problems.

We start with the LFFOTE tree pattern matching problem, the simplest of the four problems we aim to solve in this chapter. In this problem, we consider patterns to be linear and fixed. Such a pattern matches the input tree if there exists a corresponding isomorphic rootpath in the input tree.

▶ **Problem 6.1** (LFFOTE tree pattern matching problem)**.** Let $\mathcal{T}$ and $\mathcal{P}$ be ordered labeled trees. Furthermore, let $\mathcal{P}$ be linear. The *LFFOTE tree pattern matching problem* is to return the set of all nodes $v \in V(\mathcal{T})$ such that $\mathcal{P} \simeq \mathcal{T}|_v$.

In other words, a node $v \in V(\mathcal{T})$ is included in the output of the LFFOTE tree pattern matching problem if and only if the pattern tree and the rootpath $\mathcal{T}|_v$ are isomorphic trees. For example, consider the pattern trees $\mathcal{P}_1, \mathcal{P}_2$, and $\mathcal{P}_3$ illustrated in Figure 6.2 and the input tree $\mathcal{T}$ illustrated in Figure 6.1. There are two occurrences of $\mathcal{P}_1$ in $\mathcal{T}$ since $\mathcal{P} \simeq \mathcal{T}|_{\mathtt{a}^3}$ and $\mathcal{P} \simeq \mathcal{T}|_{\mathtt{a}^{10}}$. For pattern tree $\mathcal{P}_2$, there is one occurrence since $\mathcal{P}_2 \simeq \mathcal{T}|_{\mathtt{b}^{12}}$. Finally, there are two occurrences of $\mathcal{P}_3$ in $\mathcal{T}$ since $\mathcal{P}_3 \simeq \mathcal{T}|_{\mathtt{b}^6}$ and $\mathcal{P}_3 \simeq \mathcal{T}|_{\mathtt{b}^7}$.

In the context of XML and XPath, the LFFOTE tree pattern matching problem corresponds to evaluating XPath(/) queries; that is, queries that consist only of the /-axis and the node test

for node labels. For example, the pattern trees $\mathcal{P}_1, \mathcal{P}_2$, and $\mathcal{P}_3$ illustrated in Figure 6.2 can be seen as queries /a/a/a, /a/b, and /a/a/b, respectively.

The next problem we aim to solve in this chapter is similar to the one we just defined. However, we change the way of matching, so instead of matching patterns to the top-down subtrees of the input tree, we match them to its subtrees.

▶ **Problem 6.2** (LFFOSE tree pattern matching problem). Let $\mathcal{T}$ and $\mathcal{P}$ be ordered labeled trees. Furthermore, let $\mathcal{P}$ be linear. The *LFFOSE tree pattern matching problem* is to return the set of all nodes $v \in V(\mathcal{T})$ for which there exists a node $u \in V(\mathcal{T})$ such that $\mathcal{P} \simeq \mathcal{T}|_v^u$.

In other words, we look for the subtrees in the input tree that are isomorphic to a given pattern tree, and since the pattern tree is linear, we are interested only in linear subtrees of the input tree. That is, we are interested in its paths. As the output of the LFFOSE tree pattern matching problem, we return nodes such that each of them corresponds to the leaf of an isomorphic path. For example, consider the pattern trees $\mathcal{P}_1, \mathcal{P}_2$, and $\mathcal{P}_3$ illustrated in Figure 6.2 and the input tree $\mathcal{T}$ illustrated in Figure 6.1. There are two occurrences of $\mathcal{P}_1$ and $\mathcal{P}_3$ in $\mathcal{T}$ for similar reasons as we explained in the context of the LFFOTE tree pattern matching problem. However, $\mathcal{P}_2$ occurs not only at node $\mathtt{b}^{12}$, but also at nodes $\mathtt{b}^6$ and $\mathtt{b}^7$ since $\mathcal{P}_2 \simeq \mathcal{T}|_{\mathtt{b}^6}^{\mathtt{a}^5}$ and $\mathcal{P}_2 \simeq \mathcal{T}|_{\mathtt{b}^7}^{\mathtt{a}^5}$. As the example suggests, there is a relationship between the LFFOTE problem and the LFFOSE problem. For a pattern tree $\mathcal{P}$ and an input tree $\mathcal{T}$, each node included in the output of the former problem signals the occurrence of $\mathcal{P}$ in $\mathcal{T}$ for the latter problem. This is because every top-down subtree is a subtree.

Again, we can look at the LFFOSE tree pattern matching problem in the context of XML and XPath. In this case, we are interested in evaluating a set of queries that is a subset of XPath(/, //) queries. Specifically, given an alphabet Σ representing all possible element names in an XML document, the LFFOSE tree pattern matching problem corresponds to the evaluation of XPath queries generated by the following regular grammar:

$$\big(\{S, A, B\}, \Sigma \cup \{/, //\}, \{S \to //A\} \cup \{A \to a \mid aB : a \in \Sigma\} \cup \{B \to /A\}, S\big).$$

In other words, we consider queries that start with //-axis and continue only with /-axis. For example, the pattern trees $\mathcal{P}_1, \mathcal{P}_2$, and $\mathcal{P}_3$ illustrated in Figure 6.2 can be seen as queries //a/a/a, //a/b, and //a/a/b, respectively.

The next problem we consider in this chapter allows patterns to contain path wildcards. We recall that function $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true if and only if $\mathcal{T}|_v$ is isomorphic to some extension of $\mathcal{P}$, which is obtained by substituting each occurrence of path wildcard for appropriate path in of $\mathcal{T}$ or the empty tree. See Definition 3.6 and Definition 2.28.

▶ **Problem 6.3** (L(PW)FOTE tree pattern matching problem). Let $\mathcal{T}$ be an ordered labeled tree. Let $\mathcal{P}$ be a linear gapped tree. The *L(PW)FOTE tree pattern matching problem* is to return the set of all nodes $v \in V(\mathcal{T})$ such that $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true.

For example, consider the linear gapped pattern trees $\mathcal{P}_4, \mathcal{P}_5$ and $\mathcal{P}_6$ in Figure 6.3. There is one occurrence of $\mathcal{P}_4$ in $\mathcal{T}$ at node $\mathtt{c}^9$ because if the path wildcard is substituted for a linear tree of size one whose root is labeled by a, then $\mathcal{P}_4$ isomorphic to $\mathcal{T}|_{\mathtt{c}^9}$. Next, there are five occurrences of $\mathcal{P}_5$ in $\mathcal{T}$ at nodes $\mathtt{a}^2, \mathtt{a}^3, \mathtt{a}^5, \mathtt{a}^8$, and $\mathtt{a}^{10}$. Both $\mathsf{PWMatch}(\mathcal{P}_5, \mathcal{T}|_{\mathtt{a}^2})$ and $\mathsf{PWMatch}(\mathcal{P}_5, \mathcal{T}|_{\mathtt{a}^5})$ are true because if the path wildcard in $\mathcal{P}_5$ is substituted for the empty tree, then $\mathcal{P}_5$ becomes isomorphic to both $\mathcal{T}|_{\mathtt{a}^2}$ and $\mathcal{T}|_{\mathtt{a}^5}$. As for the remaining three occurrences, both $\mathsf{PWMatch}(\mathcal{P}_5, \mathcal{T}|_{\mathtt{a}^3})$ and $\mathsf{PWMatch}(\mathcal{P}_5, \mathcal{T}|_{\mathtt{a}^{10}})$ are true because $\mathcal{P}_5[\mathtt{a}]^\wr \simeq \mathcal{T}|_{\mathtt{a}^3} \simeq \mathcal{T}|_{\mathtt{a}^{10}}$, and $\mathsf{PWMatch}(\mathcal{P}_5, \mathcal{T}|_{\mathtt{a}^8})$ is true because $\mathcal{P}_5[\mathtt{ab}]^\wr \simeq \mathcal{T}|_{\mathtt{a}^8}$. Finally, $\mathcal{P}_6$ matches the input tree at node $\mathtt{c}^9$ since $\mathcal{P}_6[\mathtt{aa}, \mathtt{a}]^\wr \simeq \mathcal{T}|_{\mathtt{c}^9}$.

In the context of XML and XPath, we can see linear gapped pattern trees as XPath(/, //) queries. For example, the pattern trees $\mathcal{P}_4, \mathcal{P}_5$, and $\mathcal{P}_6$ illustrated in Figure 6.3 can be seen as queries /a/a/b//c, /a//a, and //b//c, respectively.

Note that if a given pattern tree does not contain any path wildcard, then the output of the L(PW)FOTE tree pattern matching problem is the same as the output of the LFFOTE tree

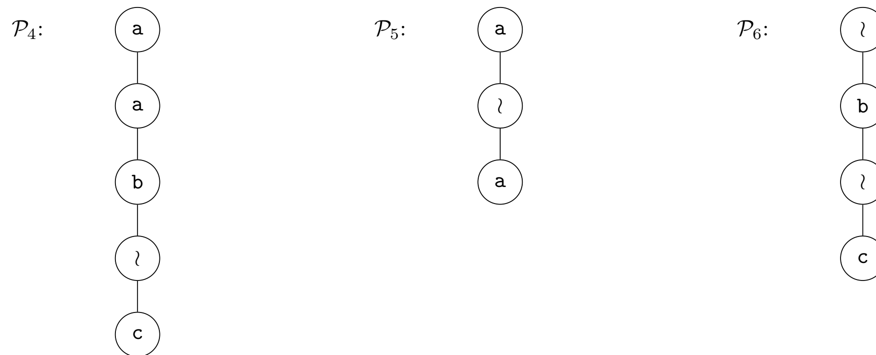


Figure 6.3 Linear gapped (pattern) trees over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \cup \{\wr\}$. Note that tree $\mathcal{P}_6$ is fully-gapped.

pattern matching problem. Moreover, suppose that a given pattern tree contains only one path wildcard in its root. In that case, the output of the L(PW)FOTE tree pattern matching problem is the same as the output of the LFFOSE tree pattern matching if we consider the input of the latter problem to be the pattern tree from the former without the path wildcard.

The last problem we define is a variant of the L(PW)FOTE tree pattern matching problem where every pattern tree is assumed to be linear and *fully-gapped*. A linear fully-gapped tree is a linear gapped tree with the maximum number of path wildcards. An example of a linear fully-gapped tree is the tree $\mathcal{P}_6$ in Figure 6.3. In the context of XML and XPath, linear fully-gapped trees correspond to XPath(//) queries.

▶ **Problem 6.4** (L(PW)FOTE tree pattern matching problem for linear fully-gapped pattern trees)**.** Let $\mathcal{T}$ be an ordered labeled tree. Let $\mathcal{P}$ be a linear fully-gapped tree. The *L(PW)FOTE tree pattern matching problem for linear fully-gapped pattern trees* is to return the set of all nodes $v \in V(\mathcal{T})$ such that $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true.

In the rest of this chapter, we present our automata-based solutions to these problems. Specifically, we present indexing methods based on string automata that can be used for the input tree to speed up the searching phase.

6.2 Automata approach

This section contains a general overview of our automata-based approach to problems defined in the previous section. In order to solve these problems with string automata, we need to encode trees as strings. We start this section by discussing a convenient linear tree notation. Then, we describe string versions of problems defined in the previous section and conclude by presenting the main ideas behind our automata-based approach.

When representing trees as strings, we always want to choose a linear tree notation suitable for the problems we aim to solve. For example, it is now not convenient to represent trees using the prefix bar notation as we did in the previous chapter. This is because we are now not interested in the bottom-up subtrees of the input tree but in matching pattern trees on the (top-down) subtrees of the input tree. Since not every (top-down) subtree in the prefix bar notation is a substring of the prefix bar notation of the input tree, solving our problems with this notation would be unnecessarily complicated.

To choose a convenient linear tree notation, we use the fact that our pattern trees are linear. Therefore, matching to top-down subtrees corresponds to matching to rootpaths, and matching to subtrees corresponds to matching to paths. In other words, it seems helpful to decompose the input tree into its (root)paths and match our patterns on them. Moreover, since both rootpaths and paths are parts of stringpaths, we can decompose the input tree into stringpaths. Thus, in

this chapter, we represent trees using the path notation described in Section 2.4.2. Moreover, we use the fact that all pattern trees are linear, and for them, we use the simpler variant of the path notation called the simple path notation. This notation was also described in Section 2.4.2.

The next step is to determine the relationship between linear pattern trees in the simple path notation and an input tree in the path notation. Specifically, we need to examine what it means for a pattern tree to match a (top-down) subtree when we represent them as strings using the notation mentioned above.

First, we consider linear fixed patterns such as those illustrated in Figure 6.2. We can see the process of matching a given linear fixed pattern to the top-down subtrees (rootpaths) of the input tree as matching the simple path notation of the pattern tree to "prefixes" of strings in the path notation of the input tree. Specifically, if a pattern tree $\mathcal{P}$ over alphabet Σ matches a rootpath in the input tree $\mathcal{T}$ over Σ that is a part of a stringpath $\mathcal{S}$, then $\mathsf{simplePath}(\mathcal{P})$ matches a prefix of $\boldsymbol{t} \downarrow \Sigma$ (recall Definition 2.1), where $\boldsymbol{t} \in \mathsf{path}(\mathcal{T})$ represents the stringpath $\mathcal{S}$. For example, consider the pattern tree $\mathcal{P}_1$ illustrated in Figure 6.2 whose simple path notation is aaa. Let $\mathcal{T}$ be the input tree illustrated in Figure 6.1. It holds that $\mathsf{path}(\mathcal{T}) = \{\mathtt{a1a1a1c}, \mathtt{a2a1b}, \mathtt{a2a2b1a1c}, \mathtt{a2a3a1c}, \mathtt{a3b1b1b}\}$. There are two occurrences of $\mathcal{P}_1$ in $\mathcal{T}$ because aaa is a prefix of $\mathtt{a1a1a1c} \downarrow \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $\mathtt{a2a3a1c} \downarrow \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. In other words, Problem 6.1 for trees in the path notation corresponds to matching patterns on "prefixes" of strings in the path notation of the input tree.

Now, consider that instead of matching a linear fixed pattern to the top-down subtrees of the input tree, we match it to its subtrees (paths). This problem corresponds to matching the simple path notation of the pattern to "substrings" of strings in the path notation of the input tree. Precisely, if a pattern tree $\mathcal{P}$ matches a path in the input tree $\mathcal{T}$ that is a part of a stringpath $\mathcal{S}$, then $\mathsf{simplePath}(\mathcal{P})$ matches a substring of $\boldsymbol{t} \downarrow \Sigma$, where $\boldsymbol{t} \in \mathsf{path}(\mathcal{T})$ represents the stringpath $\mathcal{S}$. For example, there are three occurrences of the pattern tree $\mathcal{P}_2$ illustrated in Figure 6.2 in the input tree illustrated in Figure 6.1 because $\mathsf{simplePath}(\mathcal{P}) = \mathsf{ab}$ is a substring of strings $\mathtt{a2a1b} \downarrow \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\mathtt{a2a2b1a1c} \downarrow \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and $\mathtt{a3b1b1b} \downarrow \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. In other words, Problem 6.2 for trees in the path notation corresponds to matching patterns on "substrings" of strings in the path notation of the input tree.

In Problems 6.3 and 6.4, pattern trees are allowed to contain path wildcards. First, we examine what it means for a linear fully-gapped pattern tree to match a (top-down) subtree of the input tree when trees are represented using the path notation. Given a linear fully-gapped pattern tree $\mathcal{P}$ over $\Sigma \cup \{\wr\}$, the length of its simple path notation is always even. Furthermore, each odd position contains the path wildcard, and each even position contains a symbol from Σ. This property can be used to simplify the encoding of the pattern. We can omit the path wildcards in its string representation and only keeping them in mind during the searching phase. If $\mathcal{P}$ matches a top-down subtree (rootpath) in a given input tree $\mathcal{T}$ that is a part of a stringpath $\mathcal{S}$, then $\mathsf{simplePath}(\mathcal{P}) \downarrow \Sigma$ matches a subsequence of $\boldsymbol{t} \downarrow \Sigma$, where $\boldsymbol{t} \in \mathsf{path}(\mathcal{T})$ represents the stringpath $\mathcal{S}$. For example, consider the pattern tree $\mathcal{P}_6$ illustrated in Figure 6.3, where $\mathsf{simplePath}(\mathcal{P}_6) = \wr\mathsf{b}\wr\mathsf{c}$. There is one occurrence of $\mathcal{P}_6$ in the input tree illustrated in Figure 6.1 since $\mathsf{simplePath}(\mathcal{P}_6) \downarrow \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ is a subsequence of $\mathtt{a2a2b1a1c} \downarrow \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. In other words, Problem 6.4 for trees in the path notation corresponds to matching patterns on "subsequences" of strings in the path notation of the input tree.

Finally, we examine Problem 6.3, where pattern trees are also allowed to contain path wildcards, but they do not have to be fully-gapped. In this case, we cannot ignore path wildcards in the string representation of the pattern. A path wildcard is during the searching interpreted as an unbounded gap; it can match an arbitrary substring of any length (even zero). If a gapped pattern tree $\mathcal{P}$ matches a top-down subtree (rootpath) in a given input tree $\mathcal{T}$ that is a part of a stringpath $\mathcal{S}$, then $\mathsf{simplePath}(\mathcal{P})$ matches a prefix of $\boldsymbol{t} \downarrow \Sigma$, where $\boldsymbol{t} \in \mathsf{path}(\mathcal{T})$ represents the stringpath $\mathcal{S}$. For example, there is one occurrence of pattern tree $\mathcal{P}_4$ illustrated in Figure 6.3, where $\mathsf{simplePath}(\mathcal{P}_4) = \mathsf{aab}\wr\mathsf{c}$ in the input tree illustrated in Figure 6.1 because $\mathsf{aab}\wr\mathsf{c}$ matches a prefix of $\mathtt{a2a2b1a1c} \downarrow \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ (the path wildcard is substituted by string a). In other words,

Problem 6.3 corresponds to the problem of matching patterns with unbounded gaps on prefixes of strings in the path notation of the input tree.

We are ready to describe string versions of the four problems defined in the previous section. Since we focus on offline variants of these problems, we formulate them as indexing problems. We assume that strings in the path notation of the input tree are numbered. Recall that we use $L(\Sigma, k)$ to denote a string language over $\Sigma_{\leq k}$, which is the path notation of an ordered labeled tree over Σ.

▶ **Problem 6.5** (Rootpath indexing problem for ordered labeled trees in the path notation)**.** Let Σ be an alphabet. Let $L(\Sigma, k) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|L(\Sigma,k)|}\}$. The *roothpath indexing problem for ordered labeled trees in the path notation* is to build an indexing structure for $L(\Sigma, k)$ that can be queried as follows: given a nonempty string $\boldsymbol{p}$ over Σ, return the set of all pairs (i, j) such that $\boldsymbol{p}$ is a prefix of $\boldsymbol{t}^i \downarrow \Sigma$ that ends at position j in $\boldsymbol{t}^i \downarrow \Sigma$.

▶ **Problem 6.6** (Path indexing problem for ordered labeled trees in the path notation)**.** Let Σ be an alphabet. Let $L(\Sigma, k) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|L(\Sigma,k)|}\}$. The *path indexing problem for ordered labeled trees in the path notation* is to build an indexing structure for $L(\Sigma, k)$ that can be queried as follows: given a nonempty string $\boldsymbol{p}$ over Σ, return the set of all pairs (i, j) such that $\boldsymbol{p}$ is a substring of $\boldsymbol{t}^i \downarrow \Sigma$ that ends at position j in $\boldsymbol{t}^i \downarrow \Sigma$.

▶ **Problem 6.7** (Indexing problem for ordered labeled trees in the path notation and linear fully-gapped pattern trees)**.** Let Σ be an alphabet. Let $L(\Sigma, k) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|L(\Sigma,k)|}\}$. The *indexing problem for ordered labeled trees in the path notation and linear fully-gapped pattern trees* is to build an indexing structure for $L(\Sigma, k)$ that can be queried as follows: given a nonempty string $\boldsymbol{p}$ over Σ, return the set of all pairs (i, j) such that $\boldsymbol{p}$ is a subsequence of $\boldsymbol{t}^i \downarrow \Sigma$ that ends at position j in $\boldsymbol{t}^i \downarrow \Sigma$.

▶ **Problem 6.8** (Indexing problem for ordered labeled trees in the path notation and linear gapped pattern trees)**.** Let Σ be an alphabet. Let $L(\Sigma, k) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|L(\Sigma,k)|}\}$. The *indexing problem for ordered labeled trees in the path notation and linear gapped pattern trees* is to build an indexing structure for $L(\Sigma, k)$ that can be queried as follows: given a nonempty string $\boldsymbol{p}$ over $\Sigma \cup \{\wr\}$ that is the simple path notation of a linear gapped tree over $\Sigma \cup \{\wr\}$, return the set of all pairs (i, j) such that $\boldsymbol{p}$ matches $\boldsymbol{t}^i_{1\ldots j} \downarrow \Sigma$. During the matching, symbol $\wr$ is interpreted as an unbounded gap that matches any substring of $\boldsymbol{t}^i \downarrow \Sigma$ of any length (even zero).

The output of each of the problems mentioned above is a set of pairs (i, j), where i denotes an occurrence in the i-th stringpath and j points us to the end position of the occurrence. However, the end position corresponds to a position in the stringpath projected into (sub)alphabet Σ. The position in the "original" stringpath can be computed by adding $(j - 1)$ to j since there are $j - 1$ numbers in each stringpath preceding the symbol at position j. Moreover, suppose that to each non-numerical symbol in every stringpath, we assign a link to the corresponding node. In that case, we can return nodes that represent the end of the occurrences instead of pairs (i, j).

▶ **Lemma 6.9.** *Let Σ be an alphabet. Let $L(\Sigma, k) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|L(\Sigma,k)|}\}$. Let $\boldsymbol{p}$ be a nonempty string over Σ (or $\Sigma \cup \{\wr\}$). If $\boldsymbol{p}$ occurs at $\boldsymbol{t}^i \downarrow \Sigma$ with end position j, then the corresponding end position in $\boldsymbol{t}^i$ is $j + (j - 1)$.*

Proof. This follows from the definition of the path notation; see Definition 2.44. ◀

In the rest of this section, we describe the main ideas of our automata-based solutions to the problems defined above. First, we explain the construction process. Then, we describe how our automata-based indexes can be used in the searching phase.

Since every ordered labeled tree has a finite number of rootpaths and paths, it follows that the indexing structure for Problems 6.5 and 6.6 can be based on a finite automaton. Similarly, only a finite number of linear (fully-)gapped pattern trees match a given input tree. That is,

automata-based indexes for Problems 6.7 and 6.8 have to accept a finite number of strings. Thus, these indexes can be implemented as finite automata as well.

Let $\mathcal{T}$ be an ordered labeled tree over alphabet Σ. Given $\mathsf{path}(\mathcal{T})$, we construct a deterministic finite automaton for every string $\boldsymbol{t} \downarrow \Sigma$, where $\boldsymbol{t} \in \mathsf{path}(\mathcal{T})$ such that the automaton accepts the desired types of patterns. For example, consider Problem 6.5. In this problem, we are interested in prefixes of strings from $\mathsf{path}(\mathcal{T})$. Therefore, we create a deterministic prefix automaton for each string $\boldsymbol{t} \downarrow \Sigma$, where $\boldsymbol{t} \in \mathsf{path}(\mathcal{T})$. These automata can then be combined using the product construction (see Algorithm 2.10) to obtain a deterministic finite automaton that can be used as a full index of all the rootpaths of $\mathcal{T}$. The advantage of this approach is the fast searching phase. If the time necessary to compute a transition from each state is constant, then all pattern occurrences are located in time linear to the size of the pattern. The disadvantage can be, in some cases, the space complexity of the index. Alternatively, all deterministic automata can be traversed simultaneously instead of using the product construction. This approach saves us some space at the cost of worse time complexity for searching, which is $\mathcal{O}(|\,\mathsf{leaves}(\mathcal{T})| \cdot |\mathcal{P}|)$, where $\mathcal{T}$ is the indexed tree and $\mathcal{P}$ is the pattern tree. However, in parallel systems, each automaton can be handled by a different computing node.

Our automata-based indexes can be queried as follows: Given the simple path notation $\mathsf{simplePath}(\mathcal{P})$ of a linear pattern tree $\mathcal{P}$, the automaton is run for $\mathsf{simplePath}(\mathcal{P})$. If there is an occurrence of $\mathcal{P}$, then the automaton accepts $\mathsf{simplePath}(\mathcal{P})$; otherwise it does not. Our automata-based indexes can also give the locations of occurrences that can be obtained by investigating the label of the accepting state.

In the following sections, we present details about constructing our automata-based indexes. We assume that the ordered labeled tree that is being indexed is unranked. However, in the last section of this chapter, we briefly discuss adjusting our methods for trees that are ranked, unordered, or both.

6.3 Rootpath automaton

As a warm-up, we start with the simplest problem, Problem 6.5. Our aim is to build an automaton that serves as a full index of all the rootpaths of an ordered labeled tree. We call this index the *rootpath automaton*. The rootpath automaton can be used to solve the LFFOTE tree pattern matching problem; see Problem 6.1.

▶ **Definition 6.10** (Rootpath automaton)**.** Let $\mathcal{T}$ be an ordered labeled tree. The *rootpath automaton* for $\mathsf{path}(\mathcal{T})$ accepts all the rootpaths of $\mathcal{T}$ in the simple path notation.

As outlined in the previous section, we construct all of our automata-based indexes using the same systematic approach. Given the path notation $\mathsf{path}(\mathcal{T})$ of an ordered labeled tree $\mathcal{T}$, we first construct a deterministic finite automaton for every stringpath in $\mathsf{path}(\mathcal{T})$ taking into account the type of patterns for which we are building the index. Then, we combine these automata using the product construction. Since rootpaths in the simple path notation correspond to prefixes of stringpaths in the simple path notation, the first step is to build the prefix automaton for each stringpath. This construction is described by Algorithm 6.11. The combining step is described by Algorithm 6.12.

▶ **Algorithm 6.11** (Construction of a DFA accepting all the rootpaths of a single stringpath)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Let $\mathsf{path}(\mathcal{T}) = \left\{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|\,\mathsf{path}(\mathcal{T})|}\right\}$. Given a string $\boldsymbol{t}^i \in \mathsf{path}(\mathcal{T})$, the algorithm constructs a DFA accepting all the rootpaths in the simple path notation of the stringpath represented by $\boldsymbol{t}^i$.

1. (Define the set of states.) Every state has a label j_i, where i uniquely identifies one stringpath $\boldsymbol{t}^i$ from $\mathsf{path}(\mathcal{T})$ and $j \in \{1, \ldots, |\boldsymbol{t}^i \downarrow \Sigma|\}$ corresponds to a position within $\boldsymbol{t}^i \downarrow \Sigma$.

$$\text{Set } Q \leftarrow \{j_i : 0 \leq j \leq |\boldsymbol{t}^i \downarrow \Sigma|\}.$$

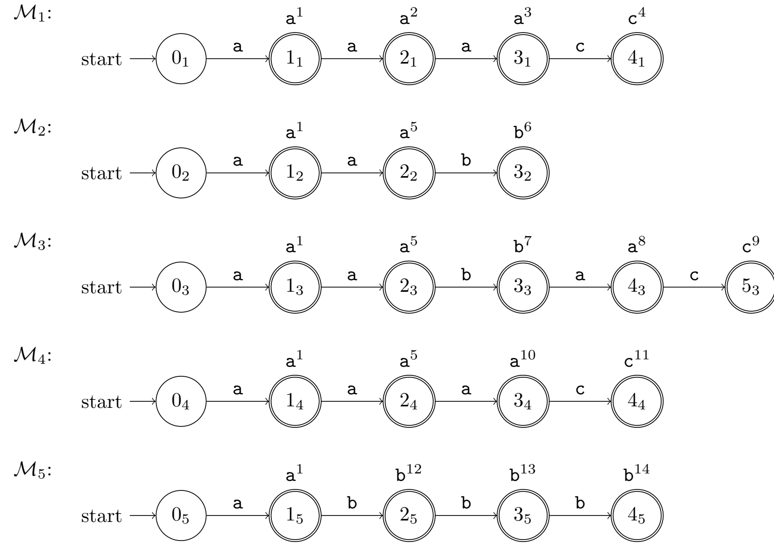


Figure 6.4 Deterministic finite automata constructed by Algorithm 6.11 for individual stringpaths of $\mathsf{path}(\mathcal{T})$, where $\mathcal{T}$ is the tree illustrated in Figure 6.1. The path notation of $\mathcal{T}$ contains following strings: $\boldsymbol{t}^1 = \texttt{a1a1a1c}$, $\boldsymbol{t}^2 = \texttt{a2a1b}$, $\boldsymbol{t}^3 = \texttt{a2a2b1a1c}$, $\boldsymbol{t}^4 = \texttt{a2a3a1c}$, and $\boldsymbol{t}^5 = \texttt{a3b1b1b}$. By accepting all nonempty prefixes of $\boldsymbol{t}^i \downarrow \{\texttt{a}, \texttt{b}, \texttt{c}\}$, automaton $\mathcal{M}_i$ accepts all the rootpaths in the simple path notation of stringpath $\boldsymbol{t}^i$.

2. (Define the start state.) Set $q_0 \leftarrow 0_i$.
3. (Define the set of final states.) Set $F \leftarrow Q \setminus \{q_0\}$.
4. (Define transitions.) Let $\boldsymbol{x} = \boldsymbol{t}^i \downarrow \Sigma$. For each position j in $\boldsymbol{x}$, set $\delta\big((j-1)_i, \boldsymbol{x}_j\big) \leftarrow j_i$.
5. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

The correctness of Algorithm 6.11 follows from two observations. First, as we discussed in Section 6.2, the simple path notation of a rootpath that is a part of stringpath $\mathcal{S}$ of an input tree $\mathcal{T}$ corresponds to a prefix of $\boldsymbol{t} \downarrow \Sigma$, where $\boldsymbol{t} \in \mathsf{path}(\mathcal{T})$ represents the stringpath $\mathcal{S}$. Second, Algorithm 6.11 is a simple modification of Algorithm 4.16.

Consider the ordered labeled tree $\mathcal{T}$ illustrated in Figure 6.1. We show the deterministic finite automata constructed by Algorithm 6.11 for individual stringpaths of $\mathsf{path}(\mathcal{T})$ in Figure 6.4.

▶ **Algorithm 6.12** (Construction of deterministic rootpath automaton)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given $\mathsf{path}(\mathcal{T})$, the algorithm constructs a deterministic rootpath automaton for $\mathsf{path}(\mathcal{T})$. The stringpaths in $\mathsf{path}(\mathcal{T})$ are assumed to be numbered from 1 to $|\,\mathsf{path}(\mathcal{T})|$. We denote the i-th stringpath of $\mathsf{path}(\mathcal{T})$ by $\boldsymbol{t}^i$.

1. (Construct automata for stringpaths.) For each stringpath $\boldsymbol{t}^i$ in $\mathsf{path}(\mathcal{T})$, build a DFA $\mathcal{M}_i$ by Algorithm 6.11.
2. (Product construction.) Combine automata $\mathcal{M}_1, \ldots, \mathcal{M}_{|\,\mathsf{path}(\mathcal{T})|}$ using Algorithm 2.10 into a deterministic finite automaton $\mathcal{M}$.
3. (Return.) Return $\mathcal{M}$.

▶ **Theorem 6.13** (Correctness of Algorithm 6.12)**.** *Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given $\mathsf{path}(\mathcal{T})$, Algorithm 6.12 constructs a deterministic rootpath automaton for $\mathsf{path}(\mathcal{T})$.*

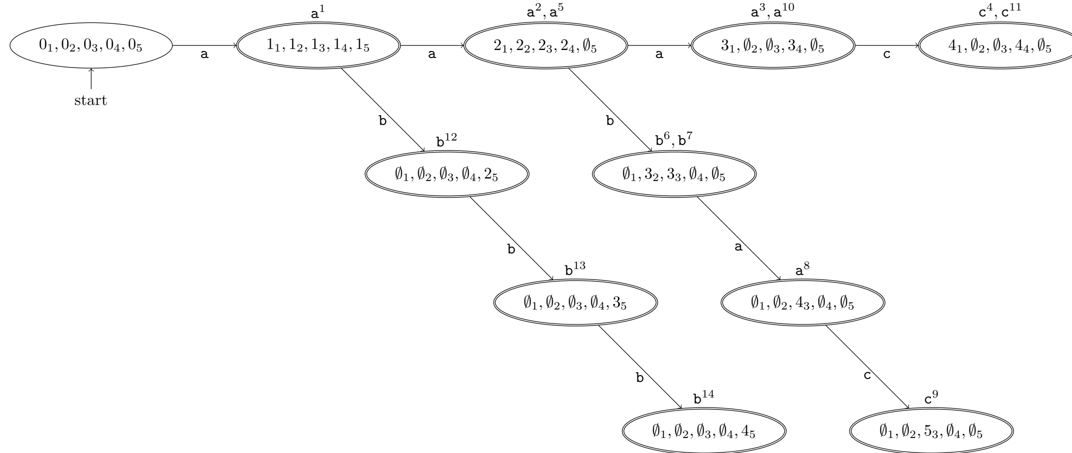


Figure 6.5 The deterministic rootpath automaton constructed by Algorithm 6.12 for the ordered labeled tree $\mathcal{T}$ illustrated in Figure 6.1, where $\mathsf{path}(\mathcal{T}) = \{\mathtt{a1a1a1c}, \mathtt{a2a1b}, \mathtt{a2a2b1a1c}, \mathtt{a2a3a1c}, \mathtt{a3b1b1b}\}$.

Proof. In Step 1, a DFA $\mathcal{M}_i$ is constructed for each stringpath $\boldsymbol{t}^i$ in $\mathsf{path}(\mathcal{T})$ so that it accepts all the rootpaths in the simple path notation of the stringpath represented by $\boldsymbol{t}^i$. In Step 2, the automata are combined so that the resulting automaton accepts language $L(\mathcal{M}_1) \cup \cdots \cup L(\mathcal{M}_{|\,\mathsf{path}(\mathcal{T})|})$. Thus, the resulting automaton accepts every rootpath of $\mathcal{T}$ in the simple path notation. Moreover, since automata $\mathcal{M}_1 \ldots, \mathcal{M}_k$ are deterministic, their combination using Algorithm 2.10 also yields a deterministic automaton. ◀

It is not hard to see that the underlying graph structure of the rootpath automaton constructed by Algorithm 6.12 is a tree. See an example in Figure 6.5. Every state is a quintuple since we combined five automata using the product construction. Every i-th element in a quintuple in a final state reports a possible occurrence of a pattern. For example, the pattern $\mathtt{aaa}$ is accepted by the automaton in the final state $(3_1, \emptyset_2, \emptyset_3, 3_4, \emptyset_5)$. The label 3_1 denotes an occurrence of the pattern in the first stringpath, that is $\mathtt{a1a1a1c}$, in position 5. Recall that to obtain this position, we need to add $(3-1)$ to 3; see Lemma 6.9. Similarly, label 3_4 reports an occurrence of the pattern in the fourth stringpath, that is $\mathtt{a2a3a1c}$, also in position 5. The labels $\emptyset_2, \emptyset_3$, and $\emptyset_5$ indicate that there is no occurrence of the pattern in the second, third nor fifth stringpath. The occurrence in the first and fourth stringpath corresponds to node $\mathtt{a}^3$ and $\mathtt{a}^{10}$ in the tree illustrated in Figure 6.1, respectively. However, it is not always the case that elements in a tuple report different occurrences. For example, the state $(1_1, 1_2, 1_3, 1_4, 1_5)$ reports occurrences of pattern $\mathtt{a}$ in each stringpaths in the first position. However, all these occurrences correspond to the same node, the root node $\mathtt{a}^1$. This can be addressed in the implementation, so the space complexity of individual states is lower. For example, we can label states by pointers to nodes representing the occurrences instead of numbers identifying the position in a stringpath. Then, states can be labeled by tuples of various sizes, where the size of a tuple corresponds to the number of unique occurrences.

Apart from the approach described by Algorithm 6.12, we may take a more direct approach and construct a deterministic rootpath automaton in a similar way the suffix trie is built; see Section 4.2.1.1. In other words, the deterministic rootpath automaton can be constructed by successively processing individual stringpaths and adding them to the automaton. We described the former approach because we use the same systematic approach in the following sections, where more complex automata are constructed.

▶ **Theorem 6.14** (Number of states of the deterministic rootpath automaton constructed by Algorithm 6.12)**.** *Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given* $\mathsf{path}(\mathcal{T})$*, the automaton constructed by Algorithm 6.12 for* $\mathsf{path}(\mathcal{T})$ *has at most $|\mathcal{T}|+1$ states.*

Proof. Apart from the start state, every state in the automaton is final. From Definition 6.10, it follows that every final state of the automaton corresponds to a rootpath in the simple path notation or a set of isomorphic rootpaths in the simple path notation. Each node v in $\mathcal{T}$ uniquely characterize a rootpath $\mathcal{T}|_v$. If each node of $\mathcal{T}$ has a distinct label, then no rootpaths are isomorphic. Thus, there at most $|\mathcal{T}|$ non-isomorphic rootpaths. ◀

▶ **Theorem 6.15** (Number of transitions of the deterministic rootpath automaton constructed by Algorithm 6.12)**.** *Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given* $\mathsf{path}(\mathcal{T})$*, the automaton constructed by Algorithm 6.12 for* $\mathsf{path}(\mathcal{T})$ *has at most $|\mathcal{T}|$ transitions.*

Proof. The automaton has a tree-like structure and accepts a finite language. Apart from the start state, every state has exactly one incoming transition. ◀

In this section, we showed the construction of a deterministic rootpath automaton. In the following section, we use the same systematic approach as outlined in Algorithm 6.12 in order to construct a finite automaton that represents a full index of all the paths of a given ordered labeled tree.

6.4 Path automaton

In this section, we present the *path automaton*, a finite automaton that serves as a full index of all the paths of an ordered labeled tree. This automaton can thus be used to solve LFFOSE tree pattern matching problem; see Problem 6.2.

▶ **Definition 6.16** (Path automaton)**.** Let $\mathcal{T}$ be an ordered labeled tree. The *path automaton* for $\mathsf{path}(\mathcal{T})$ accepts all the paths of $\mathcal{T}$ in the simple path notation.

We use the same systematic approach to construct the path automaton as we did in the previous section. Given the path notation $\mathsf{path}(\mathcal{T})$ of an ordered labeled tree $\mathcal{T}$, we first construct a deterministic finite automaton for every stringpath in $\mathsf{path}(\mathcal{T})$ taking into account the type of patterns for which we are building the index. Then, we combine these automata using the product construction. As we discussed in Section 6.2, paths correspond to substrings of stringpaths if we represent them as strings using the simple path notation. Thus, the first step is to construct the factor automaton for every stringpath. We describe this construction in Algorithm 6.17.

▶ **Algorithm 6.17** (Construction of an NFA accepting all the paths of a single stringpath)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Let $\mathsf{path}(\mathcal{T}) = \left\{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|\mathsf{path}(\mathcal{T})|}\right\}$. Given a string $\boldsymbol{t}^i \in \mathsf{path}(\mathcal{T})$, the algorithm constructs an NFA accepting all the paths in the simple path notation of the stringpath represented by $\boldsymbol{t}^i$.

1. (Define the set of states.) Every state has a label j_i, where i uniquely identifies one stringpath $\boldsymbol{t}^i$ from $\mathsf{path}(\mathcal{T})$ and $j \in \{1, \ldots, |\boldsymbol{t}^i \downarrow \Sigma|\}$ corresponds to a position within $\boldsymbol{t}^i \downarrow \Sigma$.

$$\text{Set } Q \leftarrow \{j_i : 0 \leq j \leq |\boldsymbol{t}^i \downarrow \Sigma|\}.$$

2. (Define the start state.) Set $q_0 \leftarrow 0_i$.
3. (Define the set of final states.) Set $F \leftarrow Q \setminus \{q_0\}$.
4. (Define transitions.) Let $\boldsymbol{x} = \boldsymbol{t}^i \downarrow \Sigma$. For each position j in $\boldsymbol{x}$:
   a. Set $\delta((j-1)_i, \boldsymbol{x}_j) \leftarrow \{j_i\}$.
   b. Set $\delta(0_i, \boldsymbol{x}_j) \leftarrow \delta(0_i, \boldsymbol{x}_j) \cup \{j_i\}$.
5. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

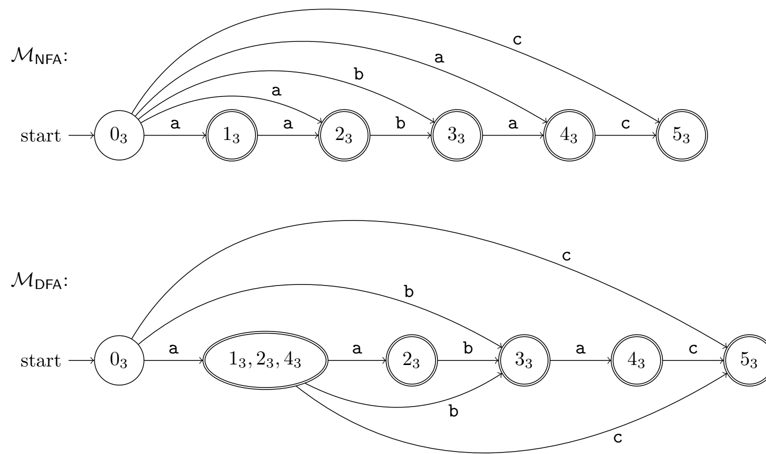


Figure 6.6 The nondeterministic finite automaton $\mathcal{M}_{\text{NFA}}$ constructed by Algorithm 6.17 for stringpath $\boldsymbol{t}^3 = \texttt{a2a2b1a1c}$ from $\texttt{path}(\mathcal{T})$, where $\mathcal{T}$ is the tree illustrated in Figure 6.1. The figure also illustrates its deterministic variant $\mathcal{M}_{\text{DFA}}$. By accepting all the nonempty substrings of $\boldsymbol{t}^3 \downarrow \{\texttt{a}, \texttt{b}, \texttt{c}\}$, both automata accept all the paths in the simple path notation of stringpath $\boldsymbol{t}^3$.

Algorithm 6.17 is a direct modification of Algorithm 4.13. The only difference is in state labeling and acceptance of the empty string; we assume the set of nodes of pattern trees to be nonempty.

The finite automaton constructed by Algorithm 6.17 is nondeterministic. To obtain its equivalent deterministic version, we can use the standard construction algorithm based on the subset construction; see Algorithm 2.6. In this way, the labels of states of the DFA represent the end positions of occurrences. For example, consider Figure 6.6, where we illustrate both the NFA constructed by Algorithm 6.17 and its deterministic variant obtained by the subset construction. Given the input a, the NFA accepts it in states 1_3, 2_3, and 4_3 which denotes that there are three occurrences of the path a in the stringpath a2a2b1a1c at position $1 + 0$, position $2 + 1$, and position $4 + 3$. The DFA gives us the same result since its states contain the corresponding states in the NFA that led to their construction.

Recall that the DFA obtained by the subset construction does not necessarily have to be minimal. For example, in Figure 6.7, we show the NFA constructed by Algorithm 6.17 for stringpath $\boldsymbol{t}^5 = \texttt{a3b1b1b}$ of the tree illustrated in Figure 6.1. The figure also shows its corresponding DFA obtained by the subset construction. The DFA is not minimal since states $\{2_5\}$ and $\{2_5, 3_5, 4_5\}$ are equivalent as well as states $\{3_5\}$ and $\{3_5, 4_5\}$. By merging the corresponding equivalent states, we save space; however, we lose the ability to correctly report end positions for some patterns. Namely, for patterns ab and abb. Thus, if we want not only to recognize all the paths in the input tree but also to correctly report the end positions of occurrences for all patterns, we cannot minimize the DFA.

The DFA can also be constructed without first constructing the NFA. However, we need to ensure that states in the DFA are labeled the same way as they would be if we created them from the NFA. Otherwise, the automaton would only be able to recognize all the paths but not return the end positions of occurrences.

We can construct the deterministic path automaton similarly as we constructed the deterministic rootpath automaton in the previous section. See Algorithm 6.18.

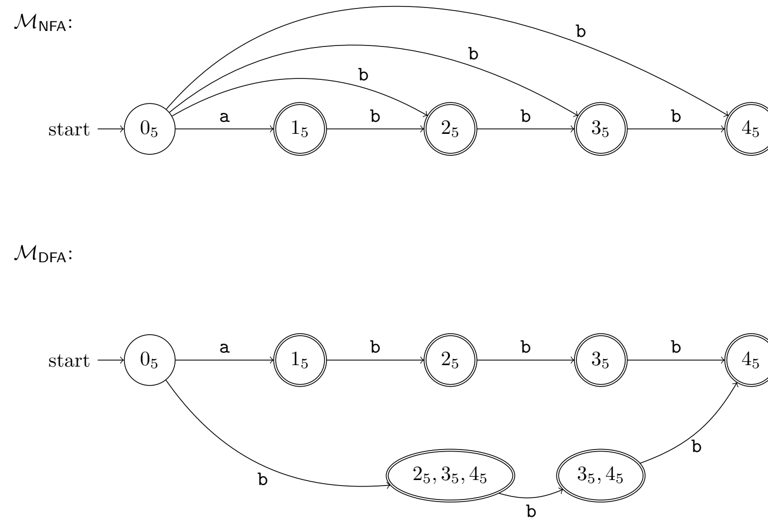


Figure 6.7 The nondeterministic finite automaton $\mathcal{M}_{\text{NFA}}$ constructed by Algorithm 6.17 for stringpath $\boldsymbol{t}^5 = \texttt{a3b1b1b}$ from $\mathsf{path}(\mathcal{T})$, where $\mathcal{T}$ is the tree illustrated in Figure 6.1. The figure also illustrates its deterministic variant $\mathcal{M}_{\text{DFA}}$.

▶ **Algorithm 6.18** (Construction of the deterministic path automaton)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given $\mathsf{path}(\mathcal{T})$, the algorithm constructs the deterministic path automaton for $\mathsf{path}(\mathcal{T})$. Stringpaths in $\mathsf{path}(\mathcal{T})$ are assumed to be numbered from 1 to $|\mathsf{path}(\mathcal{T})|$. We denote the i-th stringpath of $\mathsf{path}(\mathcal{T})$ by $\boldsymbol{t}^i$.

1. (Construct automata for stringpaths.) For each stringpath $\boldsymbol{t}^i$ in $\mathsf{path}(\mathcal{T})$, build a nondeterministic finite automaton $\mathcal{M}^i_{\text{NFA}}$ by Algorithm 6.17.
2. (Turn NFA into DFA.) For each $\mathcal{M}^i_{\text{NFA}}$, construct its equivalent deterministic variant $\mathcal{M}^i_{\text{DFA}}$ using Algorithm 2.6.
3. (Product construction.) Combine automata $\mathcal{M}^1_{\text{DFA}}, \ldots, \mathcal{M}^{|\mathsf{path}(\mathcal{T})|}_{\text{DFA}}$ using Algorithm 2.10 into a deterministic finite automaton $\mathcal{M}$.
4. (Return.) Return $\mathcal{M}$.

Algorithm 6.18 is analogous to Algorithm 6.12. Thus, its correctness can be proved using similar arguments as those used in the proof of Theorem 6.13.

Given an ordered labeled tree with l leaves, Algorithm 6.18 combines l automata in Step 3. Thus, every state of the resulting deterministic path automaton is an l-tuple. However, each element in such a tuple is a set. This is because there can be multiple occurrences of a given pattern in a single stringpath in contrast to the rootpath automaton, where the pattern can occur at most at one position in every stringpath. If the path automaton accepts a given pattern, it follows that the pattern is the simple path notation of a path in the indexed tree. We can obtain the end positions of the occurrences by investigating the label of the accepting state similarly as we did in the case of the rootpath automaton.

To prove the state complexity of the deterministic path automaton, we first need to study the state complexity of the automaton obtained by using the subset construction to the NFA constructed by Algorithm 6.17. Recall that Theorem 4.14 gives us the number of states of the minimal deterministic factor automaton. However, as we discussed, we do not want to minimize our DFA because we would lose the ability to report occurrences.

▶ **Lemma 6.19** (Number of states of the DFA constructed by Algorithm 6.17 accepting all the paths of a single stringpath). *Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ such that $|\mathcal{T}| \geq 2$. Let $\mathsf{path}(\mathcal{T}) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|\,\mathsf{path}(\mathcal{T})|}\}$. Given a string $\boldsymbol{t}^i \in \mathsf{path}(\mathcal{T})$, the DFA obtained by the subset construction from the NFA constructed by Algorithm 6.17 for $\boldsymbol{t}^i$ has at least $|\boldsymbol{t}^i \downarrow \Sigma| + 1$ states and at most $2 \cdot |\boldsymbol{t}^i \downarrow \Sigma| - 1$ states.*

Proof. Let $\mathcal{M}$ be the NFA constructed by Algorithm 6.17 for $\boldsymbol{t}^i$. Let $\mathcal{M}'$ be the nondeterministic suffix automaton constructed by Algorithm 4.9 for $|\boldsymbol{t}^i \downarrow \Sigma|$. Clearly, the underlying (unlabeled) graph structure (the set of states and transitions) of $\mathcal{M}$ is isomorphic to the graph structure of $\mathcal{M}'$; compare Algorithms 6.17 and 4.9. In other words, the two automata differ only in state labels and their sets of final states. The DFA obtained from $\mathcal{M}'$ using the subset construction is according to Theorem 4.10 minimal, which means that it has at least $|\boldsymbol{t}^i \downarrow \Sigma| + 1$ states and at most $2 \cdot |\boldsymbol{t}^i \downarrow \Sigma| - 1$ states (see Theorem 4.11). Neither the state labels nor the indication of final states affect the set of states (and transitions) created during the subset construction. Thus, the DFA obtained using the subset construction from $\mathcal{M}$ has the same underlying graph structure (the same number of states and transitions) as the DFA obtained from $\mathcal{M}'$ using the subset construction. ◀

▶ **Theorem 6.20** (Number of states of the deterministic path automaton constructed by Algorithm 6.18). *Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given $\mathsf{path}(\mathcal{T})$, the automaton constructed by Algorithm 6.18 for $\mathsf{path}(\mathcal{T})$ has $\mathcal{O}\big(\mathsf{depth}(\mathcal{T})^{|\,\mathsf{leaves}(\mathcal{T})|}\big)$ states.*

Proof. The automaton constructed by Algorithm 6.18 for $\mathsf{path}(\mathcal{T})$ is the result of product of $|\,\mathsf{path}(\mathcal{T})| = |\,\mathsf{leaves}(\mathcal{T})|$ deterministic finite automata each of which has $\mathcal{O}(\mathsf{depth}(\mathcal{T}))$ states; see Lemma 6.19. ◀

Since each state of the deterministic path automaton constructed by Algorithm 6.18 has at most $|\Sigma|$ transitions, the automaton has $\mathcal{O}\big(|\Sigma| \cdot \mathsf{depth}(\mathcal{T})^{|\,\mathsf{leaves}(\mathcal{T})|}\big)$ transitions.

In this section, we addressed the problem of path indexing for ordered labeled trees; see Problem 6.6. In the following two sections, we present automata-based indexes for linear pattern trees that are allowed to contain path wildcards.

6.5 Fully-gapped rootpath automaton

In this section, we present the *fully-gapped rootpath automaton* that is an index of an ordered labeled tree for every linear fully-gapped tree for which there exists a matching rootpath in the indexed tree. We recall that a linear fully-gapped tree is a linear gapped tree with the maximum number of path wildcards. An example of a linear fully-gapped tree is the tree $\mathcal{P}_6$ in Figure 6.3. We also recall that since path wildcards are in every linear fully-gapped tree arranged predictably, we can omit the path wildcards in its string representation and only keep them in mind during the searching phase. That is, we interpret the pattern string as a sequence of symbols rather than a compact string.

▶ **Definition 6.21** (Fully-gapped rootpath automaton). Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. The *fully-gapped rootpath automaton* for $\mathsf{path}(\mathcal{T})$ accepts every linear fully-gapped tree $\mathcal{P}$ over $\Sigma \cup \{\wr\}$ represented as string $\boldsymbol{p}$ where $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for some $v \in V(\mathcal{T})$ and $\boldsymbol{p} = \mathsf{simplePath}(\mathcal{P}) \downarrow \Sigma$.

We use the same systematic approach to build the fully-gapped rootpath automaton as we did in the previous two sections. Given the path notation $\mathsf{path}(\mathcal{T})$ of an ordered labeled tree $\mathcal{T}$, we first construct a finite automaton for every stringpath in $\mathsf{path}(\mathcal{T})$ taking into account the type of patterns for which we are building the index. See Algorithm 6.22.

▶ **Algorithm 6.22** (Construction of an NFA accepting all the linear fully-gapped trees that match a single stringpath)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Let $\mathsf{path}(\mathcal{T}) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|\,\mathsf{path}(\mathcal{T})|}\}$. Given a string $\boldsymbol{t}^i \in \mathsf{path}(\mathcal{T})$, the algorithm constructs a nondeterministic finite automaton accepting every string $\boldsymbol{p}$ such that $\boldsymbol{p} = \mathsf{simplePath}(\mathcal{P}) \downarrow \Sigma$, where $\mathcal{P}$ is a linear fully-gapped tree over $\Sigma \cup \{\wr\}$ and $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for some node v that is a part of stringpath $\boldsymbol{t}^i$.

1. (Define the set of states.) Every state has a label j_i, where i uniquely identifies one stringpath $\boldsymbol{t}^i$ from $\mathsf{path}(\mathcal{T})$ and $j \in \{1, \ldots, |\boldsymbol{t}^i \downarrow \Sigma|\}$ corresponds to a position within $\boldsymbol{t}^i \downarrow \Sigma$.

$$\text{Set } Q \leftarrow \{j_i : 0 \leq j \leq |\boldsymbol{t}^i \downarrow \Sigma|\}.$$

2. (Define the start state.) Set $q_0 \leftarrow 0_i$.

3. (Define the set of final states.) Set $F \leftarrow Q \setminus \{q_0\}$.

4. (Define transitions.) Let $\boldsymbol{x} = \boldsymbol{t}^i \downarrow \Sigma$. For each position j in $\boldsymbol{x}$:

   a. For each $k \in \{0, \ldots, j-1\}$, set $\delta(k_i, \boldsymbol{x}_j) \leftarrow \delta(k_i, \boldsymbol{x}_j) \cup \{j_i\}$.

5. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

The correctness of Algorithm 6.22 can be proved using two observations. First, as we discussed in Section 6.2, matching a linear fully-gapped pattern tree in the simple path notation from which path wildcards are removed corresponds to looking for a subsequence in stringpaths. Second, Algorithm 6.22 is a direct modification of Algorithm 4.17.

The finite automaton constructed by Algorithm 6.22 is nondeterministic. To obtain its equivalent deterministic version, we can use the standard construction algorithm based on the subset construction; see Algorithm 2.6. The labels of states of such a DFA represent the end positions of occurrences. For example, consider Figure 6.8, where we illustrate both the NFA constructed by Algorithm 6.22 and its deterministic variant obtained by the subset construction. Given the input `aa`, the NFA accepts it in states 2_3 and 4_3. Thus, there are two occurrences of the linear fully-gapped pattern tree $\wr$`a`$\wr$`a` in stringpath `a2a2b1a1c` at positions $2+1$ and $4+3$. The DFA gives us the same result.

We can construct the deterministic fully-gapped rootpath automaton in the same systematic approach as we used in the previous two sections. See Algorithm 6.23. The correctness of Algorithm 6.23 can be proved using similar arguments as those used in the proof of Theorem 6.13.

▶ **Algorithm 6.23** (Construction of the deterministic fully-gapped rootpath automaton)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given $\mathsf{path}(\mathcal{T})$, the algorithm constructs the deterministic fully-gapped rootpath automaton for $\mathsf{path}(\mathcal{T})$. Stringpaths in $\mathsf{path}(\mathcal{T})$ are assumed to be numbered from 1 to $|\,\mathsf{path}(\mathcal{T})|$. We denote the i-th stringpath of $\mathsf{path}(\mathcal{T})$ by $\boldsymbol{t}^i$.

1. (Construct automata for stringpaths.) For each stringpath $\boldsymbol{t}^i$ in $\mathsf{path}(\mathcal{T})$, build a nondeterministic finite automaton $\mathcal{M}^i_{\text{NFA}}$ by Algorithm 6.22.

2. (Turn NFA into DFA.) For each $\mathcal{M}^i_{\text{NFA}}$, construct its equivalent deterministic variant $\mathcal{M}^i_{\text{DFA}}$ using Algorithm 2.6.

3. (Product construction.) Combine automata $\mathcal{M}^1_{\text{DFA}}, \ldots, \mathcal{M}^{|\,\mathsf{path}(\mathcal{T})|}_{\text{DFA}}$ using Algorithm 2.10 into a deterministic finite automaton $\mathcal{M}$.

4. (Return.) Return $\mathcal{M}$.

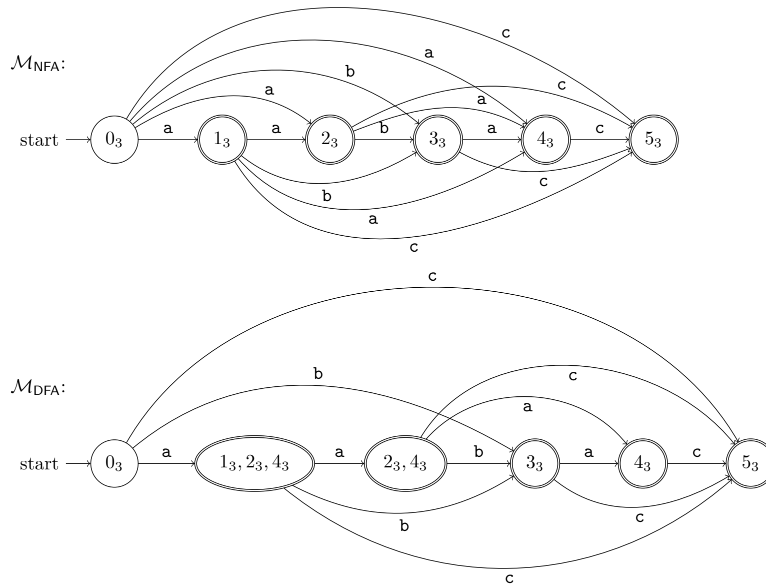


Figure 6.8 The nondeterministic finite automaton $\mathcal{M}_{\text{NFA}}$ constructed by Algorithm 6.22 for stringpath $\boldsymbol{t}^3 = \texttt{a2a2b1a1c}$ from $\mathsf{path}(\mathcal{T})$, where $\mathcal{T}$ is the tree illustrated in Figure 6.1. The figure also illustrates its deterministic variant $\mathcal{M}_{\text{DFA}}$. By accepting all nonempty subsequences of $\boldsymbol{t}^3 \downarrow \{\texttt{a}, \texttt{b}, \texttt{c}\}$, both automata accept all linear fully-gapped trees that match stringpath $\boldsymbol{t}^3$; assuming that each linear fully-gapped tree is represented in the simple path notation with path wildcards removed.

Similarly, as in the case of the path automaton, the labels of states of the automaton constructed by Algorithm 6.23 are l-tuples, where l is the number of leaves (and stringpaths) of the indexed tree. If the automaton accepts a given pattern, we can investigate the label of the accepting state to obtain the end positions of the occurrences.

In the following theorem, we discuss the state complexity of the deterministic fully-gapped rootpath automaton constructed for the path notation of an ordered labeled tree. The upper bound for the number of transitions can then be obtained by multiplying the result by the size of the alphabet.

▶ **Theorem 6.24** (Number of states of the deterministic fully-gapped rootpath automaton constructed by Algorithm 6.23)**.** *Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given $\mathsf{path}(\mathcal{T})$, the automaton constructed by Algorithm 6.23 for $\mathsf{path}(\mathcal{T})$ has $\mathcal{O}\big(\mathsf{depth}(\mathcal{T})^{|\,\mathsf{leaves}(\mathcal{T})|}\big)$ states.*

Proof. Let $\mathcal{M}$ be the NFA constructed by Algorithm 6.22 for $\boldsymbol{t}^i$. Let $\mathcal{M}'$ be the nondeterministic subsequence automaton constructed by Algorithm 4.17 for $|\boldsymbol{t}^i \downarrow \Sigma|$. Clearly, the underlying (unlabeled) graph structure (the set of states and transitions) of $\mathcal{M}$ is isomorphic to the graph structure of $\mathcal{M}'$; compare Algorithms 6.22 and 4.17. In other words, the two automata differ only in state labels and their sets of final states. According to Theorem 4.18, the DFA obtained from $\mathcal{M}'$ using the subset construction has $|\boldsymbol{t}^i \downarrow \Sigma| + 1$ states. Neither the state labels nor the indication of final states affect the set of states (and transitions) created during the subset construction. Thus, the DFA obtained using the subset construction from $\mathcal{M}$ has the same underlying graph structure (the same number of states and transitions) as the DFA obtained from $\mathcal{M}'$ using the subset construction. Therefore, the automaton constructed by Algorithm 6.23 for $\mathsf{path}(\mathcal{T})$ is the result of product of $|\,\mathsf{path}(\mathcal{T})| = |\,\mathsf{leaves}(\mathcal{T})|$ deterministic finite automata each of which has $\mathcal{O}(\mathsf{depth}(\mathcal{T}))$ states. ◀

In this section, we addressed the indexing problem for ordered labeled trees and linear fully-gapped pattern trees; see Problem 6.7. In the next section, we look at the last of our indexing problems, where pattern trees can also contain path wildcards, but they are not necessarily fully-gapped as we assumed in this section.

6.6 Gapped rootpath automaton

In this section, we address Problem 6.8 by presenting the *gapped automaton*, a finite automaton that serves as a full index of an ordered labeled tree $\mathcal{T}$ for every linear gapped tree (recall Definition 3.5) for which there exists a matching rootpath in $\mathcal{T}$. Examples of linear gapped trees are illustrated in Figure 6.3.

A linear gapped tree in the simple path notation can be seen as a string with gaps. Gaps are represented by a special symbol $\wr$, and they are unbounded. Each gap matches a string of any length (even zero).

▶ **Definition 6.25** (Gapped rootpath automaton)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. The *gapped rootpath automaton* for $\mathsf{path}(\mathcal{T})$ accepts every linear gapped tree $\mathcal{P}$ over $\Sigma \cup \{\wr\}$ in the simple path notation for which $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for some $v \in V(\mathcal{T})$.

We construct the gapped rootpath automaton as a finite automaton using the same systematic approach as we did in the previous sections. First, we introduce a method for building an automaton that accepts all linear gapped pattern trees in the simple path notation matching a rootpath of a given stringpath. See Algorithm 6.26. Given a string $\boldsymbol{t}^i \in \mathsf{path}(\mathcal{T})$, the NFA constructed by Algorithm 6.26 has $2|\boldsymbol{t}^i \downarrow \Sigma| + 1$ states.

▶ **Algorithm 6.26** (Construction of an NFA accepting all the linear gapped pattern trees that match a single stringpath)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Let $\mathsf{path}(\mathcal{T}) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \dots, \boldsymbol{t}^{|\mathsf{path}(\mathcal{T})|}\}$. Given a string $\boldsymbol{t}^i \in \mathsf{path}(\mathcal{T})$, the algorithm constructs an NFA accepting every linear gapped tree $\mathcal{P}$ over $\Sigma \cup \{\wr\}$ in the simple path notation such that $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for some node v that is a part of stringpath $\boldsymbol{t}^i$.

1. (Define the set of states.) Every state has a label j_i or j'_i, where i uniquely identifies one stringpath $\boldsymbol{t}^i$ from $\mathsf{path}(\mathcal{T})$ and $j \in \{1, \dots, |\boldsymbol{t}^i \downarrow \Sigma|\}$ corresponds to a position within $\boldsymbol{t}^i \downarrow \Sigma$.

$$\text{Set } Q \leftarrow \{j_i : 0 \leq j \leq |\boldsymbol{t}^i \downarrow \Sigma|\} \cup \{j'_i : 0 \leq j < |\boldsymbol{t}^i \downarrow \Sigma|\}.$$

2. (Define the start state.) Set $q_0 \leftarrow 0_i$.
3. (Define the set of final states.) Set $F \leftarrow \{j_i : 1 \leq j \leq |\boldsymbol{t}^i \downarrow \Sigma|\}$.
4. (Define transitions.) States labeled by j_i can be seen as states in the prefix automaton for $\boldsymbol{t}^i \downarrow \Sigma$, and states labeled by j'_i can be seen as states in the subsequence automaton for $\boldsymbol{t}^i \downarrow \Sigma$. Let $\boldsymbol{x} = \boldsymbol{t}^i \downarrow \Sigma$. For each position j in $\boldsymbol{x}$:
   a. Set $\delta((j-1)_i, \boldsymbol{x}_j) \leftarrow \{j_i\}$.
   b. Set $\delta((j-1)_i, \wr) \leftarrow \{(j-1)'_i\}$.
   c. For each $k \in \{0, \dots, j-1\}$, set $\delta(k'_i, \boldsymbol{x}_j) \leftarrow \delta(k'_i, \boldsymbol{x}_j) \cup \{j_i\}$.
5. (Return.) Return $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

The finite automaton constructed by Algorithm 6.26 can be seen as a combination of the prefix and the nondeterministic subsequence automaton. States j_i correspond to states in the prefix automaton, and states j'_i correspond to states in the subsequence automaton. The automaton can read path wildcards only from states labeled by j_i, then changes its behavior from the prefix

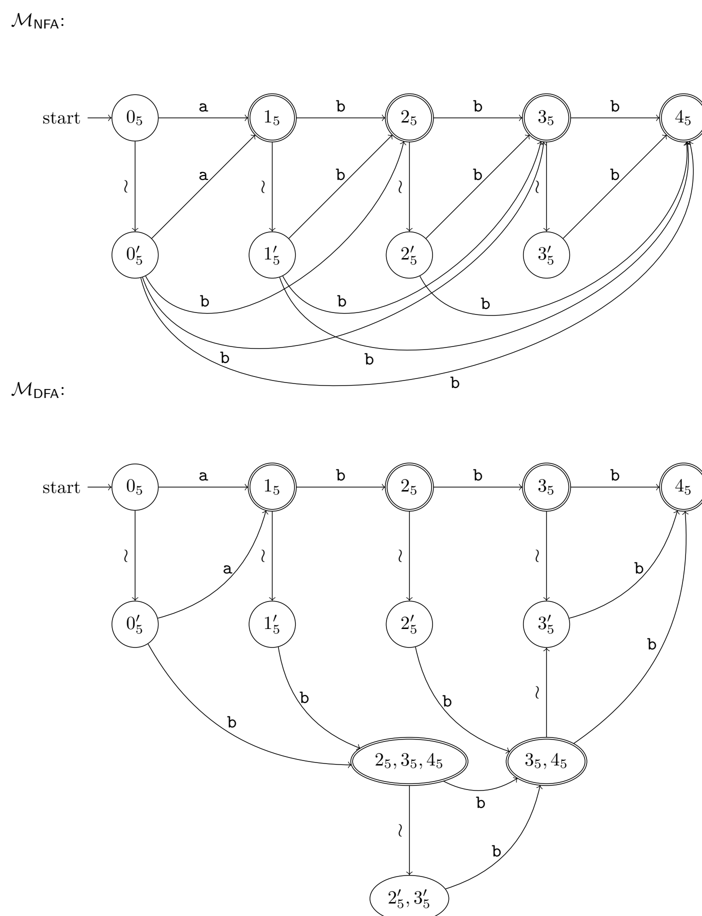


Figure 6.9 The nondeterministic finite automaton $\mathcal{M}_{\text{NFA}}$ constructed by Algorithm 6.26 for stringpath $\boldsymbol{t}^5 = \texttt{a3b1b1b}$ from $\mathsf{path}(\mathcal{T})$, where $\mathcal{T}$ is the tree illustrated in Figure 6.1. The figure also illustrates its deterministic variant $\mathcal{M}_{\text{DFA}}$. Both automata accept all linear gapped pattern trees in the simple path notation that match the stringpath $\boldsymbol{t}^5$.

to the subsequence automaton. In a state that corresponds to the subsequence automaton, we can read a symbol (not path wildcard) from the input. After that, the automaton returns to its behavior as the prefix automaton until the next path wildcard.

We show an example of the NFA constructed by Algorithm 6.26 in Figure 6.9. In this figure, we also illustrate its corresponding DFA obtained using the subset construction. Note that the DFA is not minimal since, for example, states $\{3_5\}$ and $\{3_5, 4_5\}$ are equivalent. However, as in the case of the path automaton, we do not want to minimize the DFA since we would lose the ability to report end positions for some patterns correctly.

▶ **Theorem 6.27** (Correctness of Algorithm 6.26)**.** *Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Let $\mathsf{path}(\mathcal{T}) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \ldots, \boldsymbol{t}^{|\,\mathsf{path}(\mathcal{T})|}\}$. Given a string $\boldsymbol{t}^i \in \mathsf{path}(\mathcal{T})$, the nondeterministic finite automaton constructed by Algorithm 6.26 accepts every linear gapped tree $\mathcal{P}$ over $\Sigma \cup \{\wr\}$ in the simple path notation such that $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for some node v that is a part of stringpath $\boldsymbol{t}^i$.*

Proof. The proof falls into two steps. First, we assume that $\mathcal{P}$ is a linear gapped tree over $\Sigma \cup \{\wr\}$ such that $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for some node v that is a part of stringpath $\boldsymbol{t}^i$. We show that $\mathsf{simplePath}(\mathcal{P})$ is accepted by the automaton using induction in the size of $\mathcal{P}$:

- Assume $|\mathcal{P}| = 1$. Then, $\mathcal{P}$ contains only one node, and the node is not labeled by the path wildcard. Since $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}_v)$ is true for some v that is a part of stringpath $\boldsymbol{t}^i$, we get that $\mathsf{simplePath}(\mathcal{P})$ is the prefix of $\boldsymbol{t}^i \downarrow \Sigma$ of length one. The automaton accepts this string using the transition created in Step 4a that leads from the start state 0_i to the final state 1_i.

- Assume $|\mathcal{P}| = 2$. Then, $\mathsf{simplePath}(\mathcal{P}) = \wr a$, where $a \in \Sigma$, or $\mathsf{simplePath}(\mathcal{P}) = ab$, where $a, b \in \Sigma$. In the latter case, we get that $\mathsf{simplePath}(\mathcal{P})$ is the prefix of $\boldsymbol{t}^i \downarrow \Sigma$ of length two. The automaton accepts this string using two transitions created in Step 4a: the first one leads from the start state 0_i to state 1_i and the second one leads from state 1_i to the final state 2_i. In the former case, we get that $\mathsf{simplePath}(\mathcal{P})$ is a gapped prefix of $\boldsymbol{t}^i \downarrow \Sigma$; symbol $\wr$ matches a prefix of $\boldsymbol{t}^i \downarrow \Sigma$ of any length (even zero). In the automaton this corresponds to using the transition created in Step 4b that leads from the start state 0_i to state $0'_i$. Then, from this state, the automaton can read any symbol from $\boldsymbol{t}^i \downarrow \Sigma$ using a transition created in Step 4c. Since each new transition created in Step 4c lead to a final state, the automaton accepts the string.

- Assume $|\mathcal{P}| \geq 3$ and that the claim holds for every linear gapped tree over $\Sigma \cup \{\wr\}$ of smaller size. The leaf of $\mathcal{P}$ is labeled by a symbol from Σ. The parent of the leaf can be labeled either by symbol from Σ or $\wr$. If the latter case is true, then let $\mathcal{P}'$ be the tree obtained from $\mathcal{P}$ by removing its leaf and the parent of the leaf. It is easy to check that $\mathcal{P}'$ is a linear gapped tree over $\Sigma \cup \{\wr\}$ and $\mathsf{PWMatch}(\mathcal{P}', \mathcal{T}|_u)$ is true for some node u that is a part of stringpath $\boldsymbol{t}^i$. It holds that $\mathsf{simplePath}(\mathcal{P}) = \mathsf{simplePath}(\mathcal{P}') \wr a$, where $a \in \Sigma$. From the induction hypothesis, it follows that the automaton accepts $\mathsf{simplePath}(\mathcal{P}')$, and thus ends in a final state j_i. Since the suffix of $\mathsf{simplePath}(\mathcal{P})$ is $\wr a$, the automaton can move from state j_i to state j'_i using a transition created in Step 4b and then to some final state using a transition created in Step 4c. Thus, the automaton accepts $\mathsf{simplePath}(\mathcal{P})$.

  Assume that the parent of the leaf of $\mathcal{P}$ is labeled by symbol from Σ. Then, let $\mathcal{P}'$ be the tree obtained from $\mathcal{P}$ by removing its leaf. It is easy to check that $\mathcal{P}'$ is a linear gapped tree over $\Sigma \cup \{\wr\}$ and $\mathsf{PWMatch}(\mathcal{P}', \mathcal{T}|_u)$ is true for some node u that is a part of stringpath $\boldsymbol{t}^i$. It holds that $\mathsf{simplePath}(\mathcal{P}) = \mathsf{simplePath}(\mathcal{P}')a$, where $a \in \Sigma$. From the induction hypothesis, it follows that the automaton accepts $\mathsf{simplePath}(\mathcal{P}')$, and thus ends in a final state j_i. Since the suffix of $\mathsf{simplePath}(\mathcal{P})$ is $a \in \Sigma$, the automaton moves from state j_i to state $(j+1)_i$ using a transition created in Step 4a. Thus, the automaton accepts $\mathsf{simplePath}(\mathcal{P})$.

Now, we assume that string $\boldsymbol{x}$ is accepted by the automaton. We show that $\boldsymbol{x}$ is the simple path notation of a linear gapped tree $\mathcal{P}$ over $\Sigma \cup \{\wr\}$ and $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for some node v that is a part of stringpath $\boldsymbol{t}^i$. We use induction in the length of $\boldsymbol{x}$.

- Assume $|\boldsymbol{x}| = 1$. Then, $\boldsymbol{x}$ is accepted in state 1_i using the transition created in Step 4a that leads from 0_i to 1_i. Thus, $\boldsymbol{x} = a$, where $a \in \Sigma$. This string is the simple path notation of a linear gapped tree over $\Sigma \cup \{\wr\}$ that contains only one node, and the node is labeled by a. Let $\mathcal{P}$ be that tree. Since the label of the transition that leads from 0_i to 1_i corresponds to the label of the root of stringpath $\boldsymbol{t}_i$, we get that $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for v equal to the root of stringpath $\boldsymbol{t}_i$.

- Assume $|\boldsymbol{x}| = 2$. Then, there are two possible sequences of transitions that the automaton used in order to accept $\boldsymbol{x}$. First, $\boldsymbol{x}$ can be accepted using a pair of transitions created in Step 4a that leads from 0_i to 2_i. Second, $\boldsymbol{x}$ can be accepted by using a transition created in Step 4b that leads to a state $0'_i$ followed by a transition created in Step 4c. In the first case, $\boldsymbol{x} = ab$, where $a, b \in \Sigma$. This string is the simple path notation of a linear gapped tree over $\Sigma \cup \{\wr\}$ that contains two nodes, the root labeled by a and its child labeled by b. Let $\mathcal{P}$ be that tree. Since the label of transition that leads from 0_i to 1_i and 1_i to 2_i corresponds to the label of the root of stringpath $\boldsymbol{t}_i$ and the label of the child of the root of the stringpath $\boldsymbol{t}_i$, respectively, we get that $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for v equal to the child of the root of

stringpath $\boldsymbol{t}_i$. In the second case, $\boldsymbol{x} = \wr a$, where $a \in \Sigma$. This string is the simple path notation of a linear gapped tree over $\Sigma \cup \{\wr\}$ that contains two nodes, the root labeled by path wildcard and its child labeled by a. Let $\mathcal{P}$ be that tree. Since the label of each transition that leads from $0'_i$ corresponds to the label of some node in stringpath $\boldsymbol{t}_i$, we get that $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for v equal to a node of stringpath $\boldsymbol{t}_i$ labeled by a.

- Assume $|\boldsymbol{x}| \geq 3$ and that the claim holds for shorter strings. Let j_i be the final state where the automaton accepts $\boldsymbol{x}$. It is easy to check that the last symbol of $\boldsymbol{x}$ is not path wildcard. After reading the prefix $\boldsymbol{x}_{1\dots|\boldsymbol{x}|-1}$, the automaton is either in a state $(j-1)_i$ or state k'_i, where $k \in \{0, \dots, j-1\}$:
  - In the first case, we have that $\boldsymbol{x}_{1\dots|\boldsymbol{x}|-1}$ is accepted by the automaton. Thus, by the induction hypothesis, we get that this string is the simple path notation of a linear gapped tree $\mathcal{P}$ over $\Sigma \cup \{\wr\}$ and $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for some node v that is a part of stringpath $\boldsymbol{t}^i$. Moreover, it is easy to check that v is the node of stringpath $\boldsymbol{t}_i$ whose preorder identifier is equal to $(j-1)_i$. Since there is a transition from $(j-1)_i$ to j_i labeled by $\boldsymbol{x}_{|\boldsymbol{x}|}$, we have that the child of v in stringpath $\boldsymbol{t}_i$ is labeled by $\boldsymbol{x}_{|\boldsymbol{x}|}$. Let u be that child of v. Appending the last symbol of $\boldsymbol{x}$ to $\boldsymbol{x}_{1\dots|\boldsymbol{x}|-1}$ corresponds to adding a leaf node to $\mathcal{P}$ labeled by symbol $\boldsymbol{x}_{|\boldsymbol{x}|}$. Therefore, $\boldsymbol{x}$ is the simple path notation of a linear gapped tree that matches a rootpath of stringpath $\boldsymbol{t}_i$ with the leaf node u.
  - In the second case, we have that $\boldsymbol{x}_{|\boldsymbol{x}|-1} = \wr$. It is easy to check that $\boldsymbol{x}_{|\boldsymbol{x}|-2} \in \Sigma$ and that the automaton accepts string $\boldsymbol{x}_{1\dots|\boldsymbol{x}|-2}$ in state k_i. Thus, by the induction hypothesis, we get that $\boldsymbol{x}_{1\dots|\boldsymbol{x}|-2}$ is the simple path notation of a linear gapped tree $\mathcal{P}$ over $\Sigma \cup \{\wr\}$ and $\mathsf{PWMatch}(\mathcal{P}, \mathcal{T}|_v)$ is true for some node v that is a part of stringpath $\boldsymbol{t}^i$. Moreover, it is easy to check that v is the node of stringpath $\boldsymbol{t}_i$ whose preorder identifier is equal to k. Appending the last two symbols of $\boldsymbol{x}$ to $\boldsymbol{x}_{1\dots|\boldsymbol{x}|-2}$ corresponds to inserting a tree with the root labeled by $\wr$ and its child labeled by $\boldsymbol{x}_{|\boldsymbol{x}|}$ under the leaf of $\mathcal{P}$. Since there is a transition from k'_i to j_i labeled by $\boldsymbol{x}_{|\boldsymbol{x}|}$, we have that the there is a descendant of v in stringpath $\boldsymbol{t}_i$ labeled by $\boldsymbol{x}_{|\boldsymbol{x}|}$. Let u be one of that descendants. Therefore, $\boldsymbol{x}$ is the simple path notation of a linear gapped tree that matches a rootpath of stringpath $\boldsymbol{t}_i$ with the leaf node u. ◀

We can now construct the deterministic gapped rootpath automaton by combining the automata constructed for individual stringpaths. See Algorithm 6.28. Its correctness can be proved using similar arguments as those used in the proof of Theorem 6.13.

▶ **Algorithm 6.28** (Construction of the deterministic gapped rootpath automaton)**.** Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given $\mathsf{path}(\mathcal{T})$, the algorithm constructs the deterministic gapped rootpath automaton for $\mathsf{path}(\mathcal{T})$. Stringpaths in $\mathsf{path}(\mathcal{T})$ are assumed to be numbered from 1 to $|\mathsf{path}(\mathcal{T})|$. We denote the i-th stringpath of $\mathsf{path}(\mathcal{T})$ by $\boldsymbol{t}^i$.

1. (Construct automata for stringpaths.) For each stringpath $\boldsymbol{t}^i$ in $\mathsf{path}(\mathcal{T})$, build a nondeterministic finite automaton $\mathcal{M}^i_{\mathrm{NFA}}$ by Algorithm 6.26.
2. (Turn NFA into DFA.) For each $\mathcal{M}^i_{\mathrm{NFA}}$, construct its equivalent deterministic variant $\mathcal{M}^i_{\mathrm{DFA}}$ using Algorithm 2.6.
3. (Product construction.) Combine automata $\mathcal{M}^1_{\mathrm{DFA}}, \dots, \mathcal{M}^{|\mathsf{path}(\mathcal{T})|}_{\mathrm{DFA}}$ using Algorithm 2.10 into a deterministic finite automaton $\mathcal{M}$.
4. (Return.) Return $\mathcal{M}$.

The state labels of the DFA constructed by Algorithm 6.28 are l-tuples, where l is the number of leaves of the tree that has been indexed. Similarly, as in the case of the path automaton and the fully-gapped rootpath automaton, we can investigate the label of the accepting state to obtain the end position of occurrences of a given pattern.

Before we study the state complexity of the deterministic gapped rootpath automaton for an arbitrary input tree, we first study its state complexity for a linear tree. In other words, we study the number of states of the DFA constructed for a single stringpath.

▶ **Theorem 6.29** (Number of states of the DFA accepting all the linear gapped pattern trees in the simple path notation that match a single stringpath). *Let Σ be an alphabet. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Let $\mathsf{path}(\mathcal{T}) = \{\boldsymbol{t}^1, \boldsymbol{t}^2, \dots, \boldsymbol{t}^{|\mathsf{path}(\mathcal{T})|}\}$. Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be the NFA constructed by Algorithm 6.26 for $\boldsymbol{t}^i \in \mathsf{path}(\mathcal{T})$. Then, the DFA obtained by the subset construction from $\mathcal{M}$ has $|Q'|$ states, where*

$$|Q|' = \begin{cases} 2|Q| - 3 & \text{if } |\Sigma| = 1, \\ \mathcal{O}\big(\min\big(|\Sigma|^{|Q|/(\log|\Sigma|+1)}, \sum_{a\in\Sigma}\big(2^{|Q(a)|}\big) - |\Sigma| + 1\big)\big) & \text{if } |\Sigma| \geq 2. \end{cases}$$

Proof. Let $n = |\boldsymbol{t}^i \downarrow \Sigma|$. If $|\Sigma| = 1$, then it is easy to check that the set of states of the DFA obtained by the subset construction from $\mathcal{M}$ is equal to $Q_1 \cup Q_2 \cup Q_3 \cup Q_4$, where

- $Q_1 = \{\{0\}, \{1\}, \dots, \{n\}\}$,
- $Q_2 = \{\{0'\}, \{1'\}, \dots, \{(n-1)'\}\}$,
- $Q_3 = \{\{1, 2, 3 \dots, n\}, \{2, 3 \dots, n\}, \{3 \dots, n\}, \dots, \{n-1, n\}\}$, and
- $Q_4 = \{\{1', 2', 3' \dots, n'\}, \{2', 3' \dots, n'\}, \{3 \dots, n'\}, \dots, \{(n-1)', n'\}\}$.

Clearly, $|Q_1| + |Q_2| = 2n+1 = |Q|$. Moreover, $|Q_3| = |Q_4| = n-1$. Thus, $|Q_1| + |Q_2| + |Q_3| + |Q_4| = 2|Q| - 3$.

The bound for $|\Sigma| \geq 2$ comes from the observation that $\mathcal{M}$ is a homogenous nondeterministic finite automaton that accepts a finite language. Thus, according to Theorems 2.7 and 2.9, it has $\mathcal{O}\big(|\Sigma|^{|Q|/(\log|\Sigma|+1)}\big)$ states and at most $\sum_{a\in\Sigma}\big(2^{|Q(a)|}\big) - |\Sigma| + 1$ states, respectively. ◀

▶ **Theorem 6.30** (Number of states of the deterministic gapped rootpath automaton constructed by Algorithm 6.28). *Let Σ be an alphabet such that $|\Sigma| \geq 2$. Let $\mathcal{T}$ be an ordered labeled tree over Σ. Given $\mathsf{path}(\mathcal{T})$, the automaton constructed by Algorithm 6.28 for $\mathsf{path}(\mathcal{T})$ has $\mathcal{O}\big(|Q'|^{|\mathsf{leaves}(\mathcal{T})|}\big)$ states, where $|Q'| = |\Sigma|^{|\mathsf{depth}(\mathcal{T})|/(\log|\Sigma|+1)}$.*

Proof. The automaton constructed by Algorithm 6.18 for $\mathsf{path}(\mathcal{T})$ is the result of product of $|\mathsf{path}(\mathcal{T})| = |\mathsf{leaves}(\mathcal{T})|$ deterministic finite automata each of which has $\mathcal{O}\big(|\Sigma|^{|\mathsf{depth}(\mathcal{T})|/(\log|\Sigma|+1)}\big)$ states; see Theorem 6.29. ◀

Since there are at most $|\Sigma| + 1$ outgoing transitions for each state, it follows that the upper bound for the number of transitions can be obtained by multiplying this number by the number of states.

This section addressed the indexing problem for ordered labeled trees and linear gapped pattern trees. The gapped rootpath automaton can be used to speed up the searching phase in the L(PW)FOTE tree pattern matching problem. In the following section, we summarize the results presented in this chapter.

6.7 Summary

In this chapter, we expanded arbology research by proposing a string automata approach to the problem of indexing ordered labeled trees for linear pattern trees. Specifically, given an ordered labeled tree $\mathcal{T}$, we introduced four indexing structures that are based on string automata:

rootpath automaton that is an index for all the rootpaths of $\mathcal{T}$,

path automaton that is an index for all the paths of $\mathcal{T}$,

fully-gapped rootpath automaton that is an index for every linear fully-gapped tree for which there exists a matching rootpath in $\mathcal{T}$, and

gapped rootpath automaton that is an index for every linear gapped tree for which there exists a matching rootpath in $\mathcal{T}$.

To solve tree indexing problems using string automata, we encoded trees as strings using the path notation. The convenience of this notation resides in the observation that matching linear pattern trees to rootpaths or paths of the input tree corresponds to looking for specific string parts such as prefixes, suffixes, or subsequences when trees are represented in this notation. Therefore, the principles of existing automata-based indexes for strings described in Section 4.2.1 can be used to solve the problems of tree indexing for linear patterns.

Since every ordered labeled tree has a finite number of rootpaths and paths, it follows that the structure for indexing all the rootpaths and paths can be based on a finite automaton. Similarly, only a finite number of linear (fully-)gapped pattern trees match a given input tree. Thus, indexes for these types of patterns can also be implemented as finite automata.

We used a systematic approach that involved two steps for building our automata-based indexes. First, given the path notation $\mathsf{path}(\mathcal{T})$ of an ordered labeled tree $\mathcal{T}$ over alphabet Σ, we built a deterministic finite automaton for every string $\boldsymbol{t} \downarrow \Sigma$, where $\boldsymbol{t} \in \mathsf{path}(\mathcal{T})$ such that the automaton recognizes the desired types of patterns. Then, we combined these automata using the product construction. The resulting finite automaton is deterministic and can be queried as follows: Given the simple path notation $\mathsf{simplePath}(\mathcal{P})$ of a linear pattern tree $\mathcal{P}$, the automaton is run for $\mathsf{simplePath}(\mathcal{P})$. If there is an occurrence of $\mathcal{P}$, then the automaton accepts $\mathsf{simplePath}(\mathcal{P})$; otherwise, it does not. If the time necessary to compute a transition from each state is constant, then the answer is given in time linear to the size of the pattern. Moreover, our automata-based indexes can also give the number of occurrences and the list of end positions for these occurrences. Both can be obtained by investigating the label of the accepting state.

The state complexity of the path and fully-gapped rootpath automaton is $\mathcal{O}\big(\mathsf{depth}(\mathcal{T})^{|\,\mathsf{leaves}(\mathcal{T})|}\big)$. For the rootpath automaton, we get at most $|\mathcal{T}|+1$ states and $|\mathcal{T}|$ transitions. The gapped rootpath automaton has $\mathcal{O}\big(|Q'|^{|\,\mathsf{leaves}(\mathcal{T})|}\big)$ states, where $|Q'| = |\Sigma|^{|\,\mathsf{depth}(\mathcal{T})|/(\log|\Sigma|+1)}$.

Alternatively, all deterministic automata built in the first step can be traversed simultaneously instead of combining them by the product construction. This approach saves us some space at the cost of worse time complexity for searching, which is $\mathcal{O}(|\,\mathsf{leaves}(\mathcal{T})| \cdot |\mathcal{P}|)$, where $\mathcal{T}$ is the indexed tree and $\mathcal{P}$ is the pattern tree. However, in parallel systems, each automaton can be handled by a different computing node.

In the SNINWE classification terminology introduced in Section 3.1, our automata-based indexes address four tree pattern matching problems: LFFOTE, LFFOSE, L(PW)FOTE, and a variant of the L(PW)FOTE tree pattern matching problem where pattern trees are assumed to be fully-gapped. These problems can be seen as a theoretic abstraction of evaluating XPath queries. Specifically,

- the LFFOTE tree pattern matching problem corresponds to the evaluation of XPath(/) queries that consist only of the /-axis and the node test for node labels;
- the LFFOSE tree pattern matching problem can be seen as the evaluation of a set of queries that are a subset of XPath(/, //), where queries start with //-axis and continue only with /-axis, such as query //`a`/`b`/`b`;
- the L(PW)FOTE tree pattern matching problem corresponds to the evaluation of XPath(/, //) queries; and
- the variant of L(PW)FOTE tree pattern matching problem, where pattern trees are assumed to be fully-gapped, can be seen as the evaluation of XPath(//) queries.

Therefore, our automata-based indexes can be used as auxiliary data structures that enable answering the sets of queries mentioned above for a given XML document.

From the viewpoint of database technology, the underlying idea of the rootpath automaton is the same as strong DataGuide [134] and 1-Index [135], in the sense that the index associates every distinct path from the root to the set of destination nodes. However, we believe that our description in the context of the systematic automata approach makes the ideas more comprehensible.

In this chapter, we assumed that our trees were ordered and unranked. However, it is easy to see that our automata-based indexes can also be applied to unordered trees. Since we are interested only in linear patterns, we can choose any fixed order of children and then use the methods described in this chapter. Our methods could also be applied to ranked trees. However, to look for occurrences of a linear pattern inside an input ranked tree, we may want to allow the pattern tree to not respect the rank of node labels, as linear ranked trees can only contain nodes labeled by symbols with rank 1 and one node (the leaf) with rank 0. Thus, if such a tree matches a ranked input tree, it can only match its bottom-up subtrees.

In the domain of string automata approach to indexing trees for linear patterns, we suggest the following direction for future research:

- an experimental evaluation of algorithms introduced in this chapter,
- a further analysis of the non-trivial upper bound for the number of states of the deterministic gapped rootpath automaton (using the knowledge of the NFA structure for individual stringpaths could lead to a better upper bound on the number of states of the equivalent DFA),
- an adaptation of the indexes to support more complex queries such as linear pattern trees with path variables, label wildcards, and label variables,
- exploration of possible oracle modification of the indexes, in the same way it is used for factor automaton [72] and subtree pushdown automaton [28], and
- an investigation of how the automata-based indexes introduced in this chapter can be used for computing repeats in an input tree.

Moreover, the proposed indexes are currently helpful only for non-volatile input trees, as the construction algorithms are not incremental. Therefore, future research could focus on developing iterative algorithms for the indexes.

Chapter 7

Conclusions

Motivated by intuitive and elegant solutions to various string pattern matching problems using the automata theory, this dissertation thesis explored a string automata approach to the problem of pattern matching in ordered labeled trees. Specifically, we addressed the problem of inexact tree pattern matching under the 1-degree edit distance and tree indexing for linear XPath-inspired queries. In both cases, we presented a systematic approach that is based on finite automata. We consider our results to be a part and further development of arbology research, an algorithmic discipline that focuses on the use of string automata for tree matching in the same way that string automata are used as a unifying strategy for string pattern matching.

In this chapter, we turn to the research questions formulated in Chapter 1 and discuss the answers and contributions provided by this dissertation thesis. We also list possible future work that can extend the results reported in this dissertation thesis.

7.1 Contributions of the dissertation thesis

As stated earlier, the problem of tree pattern matching can be defined as the search for all occurrences of a pattern in an input tree. In this dissertation thesis, we discussed that although various variants of this problem have been described in the literature, no unified naming standard for the problem exists. As a result, tree pattern matching problems are often known under several names, making the comparison of research results unnecessarily complex. This deficiency led to our first research question, Question 1.1.

We addressed Question 1.1 in Chapter 3, in which we proposed the SNINWE classification of tree pattern matching problems for ordered labeled trees. The classification categorizes tree pattern matching problems according to six criteria: the structure of the pattern, the nature of the pattern, the integrity of the pattern, the number of patterns, the way of matching the pattern, and the exactness of matching the pattern.

The SNINWE classification can be used to reference various tree pattern matching problems using abbreviations such as the LFFOSE problem, which corresponds to a tree pattern matching problem where

L the representation of a pattern is a Linear tree,

F all nodes in the pattern tree are labeled by specific, Fixed labels (no nodes are labeled by special symbols such as wildcards or variables),

F the Full pattern is considered during the matching,

O the pattern can be represented as One tree,

S we search for occurrences in the Subtrees of a given input tree, and

E we search for Exact occurrences.

The other two research questions, Questions 1.2 and 1.3, concerned the further development of arbology research. In particular, we explored a string automata approach to inexact tree pattern matching and XML processing. Both of these problems have been suggested as topics for future arbology research [25], [30]. From the domain of XML processing, we chose to explore the string automata approach to the problem of tree indexing for linear XPath-inspired queries.

Question 1.2 has been addressed in Chapter 5. To measure the similarity between trees, we used four variants of the 1-degree edit distance: the constrained simple 1-degree edit distance $d_{\mathrm{s}}^{\mathrm{c}}$, the constrained 1-degree edit distance d^{c}, the simple 1-degree edit distance d_{s}, and the (non-unit cost) 1-degree edit distance d. In other words, we addressed four variants of inexact tree pattern matching:

- inexact tree pattern matching under the constrained simple 1-degree edit distance,
- inexact tree pattern matching under the simple 1-degree edit distance,
- inexact tree pattern matching under the constrained 1-degree edit distance, and
- inexact tree pattern matching under the 1-degree edit distance.

In the SNINWE classification terminology, we addressed the NFFOBI tree pattern matching problem under four different similarity measures. Specifically, we focused on online searching.

To base our algorithms on string automata, we represented trees as strings using the prefix bar notation. Given a pattern tree $\mathcal{P}$ and a maximum number of allowed errors k, the main idea of our automata-based approach was to identify the pattern dictionary, the set of all strings representing the trees whose distance from $\mathcal{P}$ is at most k. Then, we built a dictionary automaton called the 1-degree matching automaton.

To find the positions of all the occurrences of the pattern tree in a given input tree $\mathcal{T}$, the 1-degree matching automaton is run on the prefix bar notation of $\mathcal{T}$. The automaton then reports a match every time it goes through a final state.

Because the pattern dictionary is always a finite language, we constructed the 1-degree matching automaton as a finite automaton. However, this means that the automaton cannot recognize whether the input string truly represents a tree. This is because such a language is not regular.

For each of the four variants of the inexact tree pattern matching problem, we presented an algorithm that constructed the 1-degree matching automaton as a finite automaton with ε-transitions. This automaton can then be transformed into an equivalent DFA. Assuming that the time necessary to compute a transition from each state is constant, the DFA can locate all pattern occurrences in time that is linear to the size of the input tree. However, this approach comes with high state complexity that limits its practicality. We proved a non-trivial upper bound on the state complexity of the 1-degree matching DFA for $d_{\mathrm{s}}^{\mathrm{c}}$, which is $\mathcal{O}\big(|\Sigma|^k \cdot k \cdot m^{k+1}\big)$, where Σ is the alphabet of all possible node labels, m is the number of nodes of the pattern tree, and k is the maximum number of errors allowed.

As an alternative approach, we presented an algorithm based on dynamic programming, the most widely used approach for computing tree edit distance and solving inexact tree pattern matching problems. However, existing methods lack clear references to a systematic approach of the standard theory of formal languages and automata. We showed that each of our dynamic programming algorithms is a simulator of the corresponding 1-degree matching ε-NFA. Given a pattern tree and an input tree with m and n nodes, respectively, our dynamic programming approach comes with $\mathcal{O}(mn)$ time complexity and $\mathcal{O}(mk)$ space complexity. For the inexact tree pattern matching problem under $d_{\mathrm{s}}^{\mathrm{c}}$, the space complexity is $\mathcal{O}(m)$.

In Chapter 6, we addressed Question 1.3 by presenting a systematic automata-based approach to tree indexing for linear XPath-inspired queries. All the queries we considered could be represented as linear trees. Therefore, the pushdown store for processing the underlying tree structure, as used in existing arbology indexes, is not needed in this case. Thus, we used a finite automaton as the computational model. Specifically, given an ordered labeled tree $\mathcal{T}$, we introduced four indexing structures that are based on finite automata:

- the rootpath automaton that is an index for all the rootpaths of $\mathcal{T}$,
- the path automaton that is an index for all the paths of $\mathcal{T}$,
- the fully-gapped rootpath automaton that is an index for every linear fully-gapped tree for which there exists a matching rootpath in $\mathcal{T}$, and
- the gapped rootpath automaton that is an index for every linear gapped tree for which there exists a matching rootpath in $\mathcal{T}$.

In the SNINWE classification terminology, our automata-based indexes address the following tree pattern matching problems: LFFOTE, LFFOSE, L(PW)FOTE, and a variant of the L(PW)FOTE tree pattern matching problem where pattern trees are assumed to be fully-gapped. These problems can be seen as a theoretic abstraction of evaluating XPath queries. Specifically,

- the LFFOTE tree pattern matching problem corresponds to the evaluation of XPath(/) queries that consist only of the /-axis and the node test for node labels;
- the LFFOSE tree pattern matching problem can be seen as the evaluation of a set of queries that are a subset of XPath(/, //), where queries start with //-axis and continue only with /-axis, such as query //`a`/`b`/`b`;
- the L(PW)FOTE tree pattern matching problem corresponds to the evaluation of XPath(/, //) queries; and
- the variant of L(PW)FOTE tree pattern matching problem, where pattern trees are assumed to be fully-gapped, can be seen as the evaluation of XPath(//) queries.

Therefore, our automata-based indexes can be used as auxiliary data structures that enable answering the sets of queries mentioned above for a given XML document.

To solve the problems of tree indexing using string automata, we encoded trees as strings using the path notation. The convenience of this notation resides in the observation that matching linear pattern trees to rootpaths or paths of the input tree corresponds to looking for specific string parts such as prefixes, suffixes, or subsequences when trees are represented in this notation. Therefore, the principles of existing automata-based indexes for strings can be used to solve the problems of tree indexing for linear patterns.

Given an ordered labeled tree $\mathcal{T}$, the searching phase uses the index built for $\mathcal{T}$, reads a linear tree query of size m, and finds the answer in time $\mathcal{O}(m)$, assuming that the time necessary to compute a transition from each state is constant. In other words, the searching time does not depend on the size of the indexed tree.

The state complexity of the path and fully-gapped rootpath automaton is $\mathcal{O}\big(\mathsf{depth}(\mathcal{T})^{|\mathsf{leaves}(\mathcal{T})|}\big)$. For the rootpath automaton, we get at most $|\mathcal{T}|+1$ states and $|\mathcal{T}|$ transitions. The gapped rootpath automaton has $\mathcal{O}\big(|Q'|^{|\mathsf{leaves}(\mathcal{T})|}\big)$ states, where $|Q'| = |\Sigma|^{|\mathsf{depth}(\mathcal{T})|/(\log|\Sigma|+1)}$.

When we addressed the tree indexing problem, we assumed that the trees were ordered and unranked. However, our automata-based indexes can also be applied to unordered trees. This is because any fixed order of children in the indexed tree can be chosen to evaluate linear patterns. Our methods can also be applied to ranked trees. However, to look for occurrences of a linear pattern inside an input ranked tree, we may want to allow the pattern tree to not respect the rank of node labels, as linear ranked trees can only contain nodes labeled by symbols with rank 1 and one node (the leaf) with rank 0. Thus, if such a tree matches a ranked input tree, it can only match its bottom-up subtrees.

7.2 Future work

In the domain of string automata approach to inexact tree pattern matching, we suggest the following direction for future research:

- an experimental evaluation of algorithms introduced in this dissertation thesis,
- an exploration of bit parallelism as a way of simulating the proposed nondeterministic finite automata,
- an investigation of the properties of the dynamic programming array and an exploration of whether the properties can be used to improve the algorithms introduced in this dissertation thesis,
- an exploration of automata approach to inexact tree pattern matching under other edit distances and for different types of trees, and
- further inspection of the space complexity of the deterministic finite automata.

Moreover, to our knowledge, there are no known solutions to the inexact tree pattern matching problems that are based on tree automata. Thus, future research can address this gap.

In the domain of string automata approach to indexing trees for linear patterns, we suggest the following direction for future research:

- an experimental evaluation of algorithms introduced in this dissertation thesis,
- a further analysis of the non-trivial upper bound for the number of states of the deterministic gapped rootpath automaton (using the knowledge of the NFA structure for individual stringpaths could lead to a better upper bound on the number of states of the equivalent DFA),
- an adaptation of the indexes to support more complex queries such as linear pattern trees with path variables, label wildcards, and label variables,
- exploration of possible oracle modification of the indexes, in the same way it is used for factor automaton [72] and subtree pushdown automaton [28], and
- an investigation of how the automata-based indexes introduced in this dissertation thesis can be used for computing repeats in an input tree.

Moreover, the proposed indexes are currently helpful only for non-volatile input trees, as the construction algorithms are not incremental. Therefore, future research could focus on developing iterative algorithms for the indexes.

Bibliography

[1] J. Zobel, *Writing for Computer Science*. Springer, London, 2014, ISBN: 9781447166382. DOI: 10.1007/978-1-4471-6639-9.

[2] B. A. Shapiro and K. Z. Zhang, "Comparing multiple RNA secondary structures using tree comparisons," *Computer applications in the biosciences: CABIOS*, vol. 6, no. 4, pp. 309–318, Oct. 1990, ISSN: 0266-7061. DOI: 10.1093/bioinformatics/6.4.309.

[3] J.-F. Dufayard, L. Duret, S. Penel, M. Gouy, F. Rechenmann, and G. Perrière, "Tree pattern matching in phylogenetic trees: Automatic search for orthologs or paralogs in homologous gene sequence databases," *Bioinformatics*, vol. 21, no. 11, pp. 2596–2603, Jun. 2005, ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bti325.

[4] A. V. Aho and M. Ganapathi, "Efficient tree pattern matching (extended abstract): An aid to code generation," in *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '85, New Orleans, Louisiana, USA: Association for Computing Machinery, Jan. 1985, pp. 334–340, ISBN: 9780897911474. DOI: 10.1145/318593.318663.

[5] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 4, pp. 491–516, Oct. 1989, ISSN: 0164-0925. DOI: 10.1145/69558.75700.

[6] C. M. Hoffmann and M. J. O'Donnell, "Pattern matching in trees," *Journal of the ACM*, vol. 29, no. 1, pp. 68–95, Jan. 1982, ISSN: 0004-5411, 1557-735X. DOI: 10.1145/322290.322295.

[7] J. Lu, T. W. Ling, Z. Bao, and C. Wang, "Extended XML tree pattern matching: Theories and algorithms," *IEEE transactions on knowledge and data engineering*, vol. 23, no. 3, pp. 402–416, Mar. 2011, ISSN: 1041-4347. DOI: 10.1109/TKDE.2010.126.

[8] M. A. Tahraoui, K. Pinel-Sauvagnat, C. Laitang, M. Boughanem, H. Kheddouci, and L. Ning, "A survey on tree matching and XML retrieval," *Computer Science Review*, vol. 8, pp. 1–23, May 2013, ISSN: 1574-0137, 1876-7745. DOI: 10.1016/j.cosrev.2013.02.001.

[9] R. Baeza-Yates, "A unified view to string matching algorithms," in *SOFSEM'96: Theory and Practice of Informatics*, Springer Berlin Heidelberg, 1996, pp. 1–15. DOI: 10.1007/BFb0037393.

[10] M. Crochemore and C. Hancart, "Automata for matching patterns," in *Handbook of Formal Languages: Volume 2. Linear Modeling: Background and Application*, G. Rozenberg and A. Salomaa, Eds., vol. 2, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 399–462, ISBN: 9783662076750. DOI: 10.1007/978-3-662-07675-0_9.

[11] M. Crochemore and W. Rytter, *Jewels of Stringology: Text Algorithms*. World Scientific Publishing, 2003, ISBN: 9789810247829. DOI: 10.1142/4838.

[12] B. Melichar, J. Holub, and T. Polcar, *Text Searching Algorithms-Volume I: Forward String Matching*. 2005. [Online]. Available: `https://www.stringology.org/athens/TextSearchingAlgorithms/tsa-lectures-1.pdf`.

[13] P. Antoniou, J. Holub, C. S. Iliopoulos, B. Melichar, and P. Peterlongo, "Finding common motifs with gaps using finite automata," in *Implementation and Application of Automata*, Springer Berlin Heidelberg, 2006, pp. 69–77. DOI: `10.1007/11812128_8`.

[14] J. Holub, "Finite automata in pattern matching," *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*, pp. 51–71, 2011. DOI: `10.1002/9780470892107.ch3`.

[15] J. Holub, "The finite automata approaches in stringology," *Kybernetika*, vol. 48, no. 3, pp. 386–401, 2012, ISSN: 0023-5954. [Online]. Available: `https://eudml.org/doc/246462`.

[16] B. Melichar, "String matching with k differences by finite automata," in *Proceedings of 13th International Conference on Pattern Recognition*, vol. 2, ieeexplore.ieee.org, Aug. 1996, 256–260 vol.2. DOI: `10.1109/ICPR.1996.546828`.

[17] J. Holub, "Simulation of nondeterministic finite automata in pattern matching," Ph.D. dissertation, Czech Technical University, Prague, Faculty of Electrical Engineering, 2000.

[18] M. Šimůnek and B. Melichar, "Borders and finite automata," in *Implementation and Application of Automata*, Springer Berlin Heidelberg, 2006, pp. 58–68. DOI: `10.1007/11812128_7`.

[19] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas, "The smallest automation recognizing the subwords of a text," *Theoretical computer science*, vol. 40, pp. 31–55, Jan. 1985, ISSN: 0304-3975. DOI: `10.1016/0304-3975(85)90157-4`.

[20] M. Crochemore, "Transducers and repetitions," *Theoretical computer science*, vol. 45, pp. 63–86, 1986, ISSN: 0304-3975. DOI: `10.1016/0304-3975(86)90041-1`.

[21] H. Comon, M. Dauchet, R. Gilleron, *et al.*, *Tree automata: Techniques and applications*. 2008. [Online]. Available: `https://hal.inria.fr/hal-03367725`.

[22] L. G. W. A. Cleophas, "Tree algorithms: Two taxonomies and a toolkit," Ph.D. dissertation, 2008, ISBN: 9789038612287. DOI: `10.6100/IR633481`.

[23] J. Janoušek, *Arbology: Algorithms on trees and pushdown automata*, Habilitation thesis, Brno University of Technology, Faculty of Information Technology, 2010.

[24] B. Melichar, "Arbology: Trees and pushdown automata," in *Language and Automata Theory and Applications*, Springer Berlin Heidelberg, 2010, pp. 32–49. DOI: `10.1007/978-3-642-13089-2_3`.

[25] B. Melichar, J. Janoušek, and T. Flouri, "Arbology: Trees and pushdown automata," *Kybernetika*, vol. 48, no. 3, pp. 402–428, 2012, ISSN: 0023-5954. [Online]. Available: `https://eudml.org/doc/246452`.

[26] J. Janoušek and B. Melichar, "On regular tree languages and deterministic pushdown automata," *Acta Informatica*, vol. 46, no. 7, p. 533, Sep. 2009, ISSN: 0001-5903, 1432-0525. DOI: `10.1007/s00236-009-0104-9`.

[27] T. Flouri, C. S. Iliopoulos, J. Janoušek, B. Melichar, and S. P. Pissis, "Tree indexing by pushdown automata and subtree repeats," in *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, ieeexplore.ieee.org, Sep. 2011, pp. 899–902.

[28] M. Plicka, J. Janoušek, and B. Melichar, "Subtree oracle pushdown automata for ranked and unranked ordered trees," in *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, ieeexplore.ieee.org, Sep. 2011, pp. 903–906.

[29] J. Trávníček, J. Janoušek, and B. Melichar, "Indexing trees by pushdown automata for nonlinear tree pattern matching," in *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, ieeexplore.ieee.org, Sep. 2011, pp. 871–878.

[30] T. Flouri, “Pattern matching in tree structures,” Ph.D. dissertation, Czech Technical University in Prague, Faculty of Information Technology, 2012.

[31] J. Trávníček, J. Janoušek, and B. Melichar, “Indexing ordered trees for (nonlinear) tree pattern matching by pushdown automata,” *Computer Science and Information Systems*, vol. 9, no. 3, pp. 1125–1153, 2012. DOI: `10.2298/CSIS111220024T`.

[32] 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*, Springer International Publishing, 2014, pp. 198–209. DOI: `10.1007/978-3-319-09704-6_18`.

[33] J. Trávníček, “(Nonlinear) tree pattern indexing and backward matching,” Ph.D. dissertation, Czech Technical University in Prague, Faculty of Information Technology, 2018.

[34] J. Janoušek, B. Melichar, and M. Poliak, “Tree compression pushdown automaton,” *Kybernetika*, vol. 48, no. 3, pp. 429–452, 2012, ISSN: 0023-5954. [Online]. Available: `https://eudml.org/doc/247233`.

[35] M. Poliak, “On indexes of ordered trees for subtrees and tree patterns and their space complexities,” Ph.D. dissertation, Czech Technical University in Prague, Faculty of Information Technology, 2017.

[36] J. Lahoda and J. Žďárek, “Simple tree pattern matching for trees in the prefix bar notation,” *Discrete applied mathematics*, vol. 163, pp. 343–351, Jan. 2014, ISSN: 0166-218X, 1872-6771. DOI: `10.1016/j.dam.2013.07.018`.

[37] P. Cserkuti, T. Levendovszky, and H. Charaf, “Survey on subtree matching,” in *2006 International Conference on Intelligent Engineering Systems*, Jun. 2006, pp. 216–221. DOI: `10.1109/INES.2006.1689372`.

[38] T. Flouri, J. Janoušek, and B. Melichar, “Subtree matching by pushdown automata,” *Computer Science and Information Systems*, vol. 7, no. 2, pp. 331–357, 2010, ISSN: 1820-0214, 2406-1018. DOI: `10.2298/csis1002331f`.

[39] G. Valiente, *Algorithms on Trees and Graphs*, second, ser. Texts in Computer Science. Springer International Publishing, 2021, ISBN: 9783030818852. DOI: `10.1007/978-3-030-81885-2`.

[40] A. Abboud, A. Backurs, T. D. Hansen, V. Vassilevska Williams, and O. Zamir, “Subtree isomorphism revisited,” *ACM Transactions on Algorithms*, vol. 14, no. 3, pp. 1–23, Jun. 2018, ISSN: 1549-6325. DOI: `10.1145/3093239`.

[41] M. Christou, T. Flouri, C. S. Iliopoulos, *et al.*, “Tree template matching in unranked ordered trees,” *Journal of discrete algorithms*, vol. 20, pp. 51–60, May 2013, ISSN: 1570-8667. DOI: `10.1016/j.jda.2013.02.001`.

[42] S. M. Selkow, “The tree-to-tree editing problem,” *Information processing letters*, vol. 6, no. 6, pp. 184–186, Dec. 1977, ISSN: 0020-0190, 1872-6119. DOI: `10.1016/0020-0190(77)90064-3`.

[43] R. A. Wagner and M. J. Fischer, “The String-to-String correction problem,” *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974, ISSN: 0004-5411. DOI: `10.1145/321796.321811`.

[44] M.-M. Deza and E. Deza, *Dictionary of Distances*. Elsevier, Nov. 2006, ISBN: 9780080465548.

[45] P. Bille, “A survey on tree edit distance and related problems,” *Theoretical computer science*, vol. 337, no. 1, pp. 217–239, Jun. 2005, ISSN: 0304-3975. DOI: `10.1016/j.tcs.2004.12.030`.

[46] E. Ukkonen, “Finding approximate patterns in strings,” *Journal of algorithms & computational technology*, vol. 6, no. 1, pp. 132–137, Mar. 1985, ISSN: 1748-3018, 0196-6774. DOI: `10.1016/0196-6774(85)90023-9`.

[47] R. Baeza-Yates and G. Navarro, “Faster approximate string matching,” *Algorithmica. An International Journal in Computer Science*, vol. 23, no. 2, pp. 127–158, Feb. 1999, ISSN: 0178-4617. DOI: 10.1007/PL00009253.

[48] S. Wu and U. Manber, “Fast text searching: Allowing errors,” *Communications of the ACM*, vol. 35, no. 10, pp. 83–91, Oct. 1992, ISSN: 0001-0782. DOI: 10.1145/135239.135244.

[49] F. Luccio and L. Pagli, “Simple solutions for approximate tree matching problems,” in *TAPSOFT ’91*, Springer Berlin Heidelberg, 1991, pp. 193–201. DOI: 10.1007/3-540-53982-4_11.

[50] J. Tsong-Li Wang, K. Zhang, K. Jeong, and D. Shasha, “A system for approximate tree matching,” *IEEE transactions on knowledge and data engineering*, vol. 6, no. 4, pp. 559–571, Aug. 1994, ISSN: 1041-4347, 1558-2191. DOI: 10.1109/69.298173.

[51] K. Z. Zhang, D. Shasha, and J. T. L. Wang, “Approximate tree matching in the presence of variable length don’t cares,” *Journal of algorithms & computational technology*, vol. 16, no. 1, pp. 33–66, Jan. 1994, ISSN: 1748-3018. DOI: 10.1006/jagm.1994.1003.

[52] F. Luccio and L. Pagli, “Approximate matching for 2 families of trees,” *Information and Computation*, vol. 123, no. 1, pp. 111–120, Nov. 1995, ISSN: 0890-5401. DOI: 10.1006/inco.1995.1160.

[53] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Pearson Education, 2014, ISBN: 9781292039053.

[54] D. C. Kozen, *Automata and Computability.* Springer Science & Business Media, Dec. 2012, ISBN: 9781461218449.

[55] M. O. Rabin and D. Scott, “Finite automata and their decision problems,” *IBM Journal of Research and Development*, vol. 3, no. 2, pp. 114–125, Apr. 1959, ISSN: 0018-8646. DOI: 10.1147/rd.32.0114.

[56] F. R. Moore, “On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata,” *IEEE transactions on computers. Institute of Electrical and Electronics Engineers*, vol. C-20, no. 10, pp. 1211–1214, Oct. 1971, ISSN: 0018-9340, 1557-9956. DOI: 10.1109/T-C.1971.223108.

[57] K. Salomaa and S. Yu, “NFA to DFA transformation for finite languages,” in *Automata Implementation*, Springer Berlin Heidelberg, 1997, pp. 149–158. DOI: 10.1007/3-540-63174-7_12.

[58] B. Melichar and J. Skryja, “On the size of deterministic finite automata,” in *Implementation and Application of Automata*, Springer Berlin Heidelberg, 2002, pp. 202–213. DOI: 10.1007/3-540-36390-4_17.

[59] J.-M. Champarnaud, D. Ziadi, and J.-L. Ponty, “Determinization of glushkov automata,” in *Automata Implementation*, Springer Berlin Heidelberg, 1999, pp. 57–68. DOI: 10.1007/3-540-48057-9_5.

[60] S. Yu, “State complexity of regular languages,” *Journal of Automata, Languages and Combinatorics*, vol. 6, no. 2, pp. 221–234, 2001, ISSN: 1430-189X. DOI: 10.25596/jalc-2001-221.

[61] J. W. Thatcher and J. B. Wright, “Generalized finite automata theory with an application to a decision problem of second-order logic,” *Mathematical Systems Theory*, vol. 2, no. 1, pp. 57–81, Mar. 1968, ISSN: 0025-5661, 1433-0490. DOI: 10.1007/bf01691346.

[62] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, third edition.* MIT Press, Jul. 2009, ISBN: 9780262033848.

[63] D. Knuth, *The art of computer programming*, 3rd ed. Reading, Mass: Addison-Wesley, 1997, ISBN: 9780201853940.

[64] S. Gorn, "Explicit definitions and linguistic dominoes," in *Proceedings of a Conference held at the University of Western Ontario September 10-11, 1965*, University of Toronto Press, 1965, pp. 77–115. DOI: 10.3138/9781487592769-008.

[65] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving, Second Edition.* Courier Dover Publications, Jun. 2015, ISBN: 9780486780825.

[66] J. Stoklasa, J. Janoušek, and B. Melichar, *Subtree pushdown automata for trees in bar notation*, London Stringology Days (2010), London, 2010.

[67] J. Cristau, C. Löding, and W. Thomas, "Deterministic automata on unranked trees," in *Fundamentals of Computation Theory*, Springer Berlin Heidelberg, 2005, pp. 68–79. DOI: 10.1007/11537311_7.

[68] R. Ramesh and I. V. Ramakrishnan, "Nonlinear pattern matching in trees," *Journal of the ACM*, vol. 39, no. 2, pp. 295–316, Apr. 1992, ISSN: 0004-5411. DOI: 10.1145/128749.128752.

[69] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, Dec. 1989, ISSN: 0097-5397. DOI: 10.1137/0218082.

[70] B. Melichar and J. Holub, *6D classification of pattern matching problems*, The Prague Stringology Club Workshop '97, Prague, 1997. [Online]. Available: https://www.stringology.org/event/1997/p3.html.

[71] M. Crochemore, C. Hancart, and T. Lecroq, *Algorithms on Strings.* Cambridge University Press, Nov. 2014, ISBN: 9781107670990. DOI: 10.1017/CBO9780511546853.

[72] C. Allauzen, M. Crochemore, and M. Raffinot, "Factor oracle: A new structure for pattern matching," in *SOFSEM'99: Theory and Practice of Informatics*, Springer Berlin Heidelberg, 1999, pp. 295–310. DOI: 10.1007/3-540-47849-3_18.

[73] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001, ISSN: 0360-0300. DOI: 10.1145/375360.375365.

[74] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, 1966, pp. 707–710. [Online]. Available: https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf.

[75] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, Apr. 1950, ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[76] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, 1964, ISSN: 0001-0782. DOI: 10.1145/363958.363994.

[77] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970, ISSN: 0022-2836. DOI: 10.1016/0022-2836(70)90057-4.

[78] A. Apostolico and C. Guerra, "The longest common subsequence problem revisited," *Algorithmica. An International Journal in Computer Science*, vol. 2, no. 1-4, pp. 315–336, Nov. 1987, ISSN: 0178-4617, 1432-0541. DOI: 10.1007/bf01840365.

[79] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen, "Episode matching," in *Combinatorial Pattern Matching*, Springer Berlin Heidelberg, 1997, pp. 12–27. DOI: 10.1007/3-540-63220-4_46.

[80] G. M. Landau and U. Vishkin, "Fast string matching with k differences," *Journal of Computer and System Sciences*, vol. 37, no. 1, pp. 63–78, Aug. 1988, ISSN: 0022-0000. DOI: 10.1016/0022-0000(88)90045-1.

[81] Y. Chen and H. H. Nguyen, "On the string matching with k differences in DNA databases," *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 14, no. 6, pp. 903–915, Feb. 2021, ISSN: 2150-8097. DOI: 10.14778/3447689.3447695.

[82] P. H. Sellers, "The theory and computation of evolutionary distances: Pattern recognition," *Journal of algorithms & computational technology*, vol. 1, no. 4, pp. 359–373, Dec. 1980, ISSN: 1748-3018, 0196-6774. DOI: 10.1016/0196-6774(80)90016-4.

[83] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM*, vol. 46, no. 3, pp. 395–415, May 1999, ISSN: 0004-5411. DOI: 10.1145/316542.316550.

[84] E. Ukkonen, "Algorithms for approximate string matching," *Information and Control*, vol. 64, no. 1, pp. 100–118, Jan. 1985, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(85)80046-2.

[85] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System Sciences*, vol. 20, no. 1, pp. 18–31, Feb. 1980, ISSN: 0022-0000. DOI: 10.1016/0022-0000(80)90002-1.

[86] A. Backurs and P. Indyk, "Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)," in *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, ser. STOC '15, Portland, Oregon, USA: Association for Computing Machinery, Jun. 2015, pp. 51–58, ISBN: 9781450335362. DOI: 10.1145/2746539.2746612.

[87] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, May 1997, ISBN: 9780521585194.

[88] B. Melichar, "Approximate string matching by finite automata," in *Computer Analysis of Images and Patterns*, Springer Berlin Heidelberg, 1995, pp. 342–349. DOI: 10.1007/3-540-60268-2_315.

[89] G. M. Landau and U. Vishkin, "Fast parallel and serial approximate string matching," *Journal of algorithms & computational technology*, vol. 10, no. 2, pp. 157–169, Jun. 1989, ISSN: 1748-3018, 0196-6774. DOI: 10.1016/0196-6774(89)90010-2.

[90] Z. Galil and K. Park, "An improved algorithm for approximate string matching," *SIAM Journal on Computing*, vol. 19, no. 6, pp. 989–999, Dec. 1990, ISSN: 0097-5397. DOI: 10.1137/0219067.

[91] G. Navarro, "A partial deterministic automaton for approximate string matching," in *In Proceedings of Fourth South American Workshop on String Processing (WSP'97)*, Carleton University Press, 1997, pp. 112–124.

[92] K. Zhang, J. T. L. Wang, and D. Shasha, "On the editing distance between undirected acyclic graphs and related problems," in *Combinatorial Pattern Matching*, Springer Berlin Heidelberg, 1995, pp. 395–407. DOI: 10.1007/3-540-60044-2_58.

[93] K.-C. Tai, "The tree-to-tree correction problem," *Journal of the ACM*, vol. 26, no. 3, pp. 422–433, Jul. 1979, ISSN: 0004-5411. DOI: 10.1145/322139.322143.

[94] S. Lu, "A tree-to-tree distance and its application to cluster analysis," *IEEE transactions on pattern analysis and machine intelligence*, vol. PAMI-1, no. 2, pp. 219–224, Apr. 1979, ISSN: 0162-8828, 1939-3539. DOI: 10.1109/TPAMI.1979.6786615.

[95] K. Zhang, "Algorithms for the constrained editing distance between ordered labeled trees and related problems," *Pattern recognition*, vol. 28, no. 3, pp. 463–474, Mar. 1995, ISSN: 0031-3203. DOI: 10.1016/0031-3203(94)00109-Y.

[96] J. T. L. Wang and K. Zhang, "Finding similar consensus between trees: An algorithm and a distance hierarchy," *Pattern recognition*, vol. 34, no. 1, pp. 127–137, Jan. 2001, ISSN: 0031-3203. DOI: 10.1016/S0031-3203(99)00199-5.

[97] T. Yoshino and K. Hirata, “Tai mapping hierarchy for rooted labeled trees through common subforest,” *Theory of Computing Systems*, vol. 60, no. 4, pp. 759–783, May 2017, ISSN: 1432-4350, 1433-0490. DOI: 10.1007/s00224-016-9705-1.

[98] D. Shasha and K. Zhang, “Fast algorithms for the unit cost editing distance between trees,” *Journal of algorithms & computational technology*, vol. 11, no. 4, pp. 581–621, Dec. 1990, ISSN: 1748-3018, 0196-6774. DOI: 10.1016/0196-6774(90)90011-3.

[99] S. Akmal and C. Jin, “Faster algorithms for bounded tree edit distance,” in *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 12:1–12:15. DOI: 10.4230/LIPIcs.ICALP.2021.12.

[100] S. Y. Lu, “A tree-matching algorithm based on node splitting and merging,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 6, no. 2, pp. 249–256, Feb. 1984, ISSN: 0162-8828, 0098-5589. DOI: 10.1109/tpami.1984.4767511.

[101] P. Klein, S. Tirthapura, D. Sharvit, and B. B. Kimia, “A tree-edit-distance algorithm for comparing simple, closed shapes,” in *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, ACM/SIAM, 2000, pp. 696–704. [Online]. Available: https://dl.acm.org/citation.cfm?id=338219.338628.

[102] K. Zhang, R. Statman, and D. Shasha, “On the editing distance between unordered labeled trees,” *Information processing letters*, vol. 42, no. 3, pp. 133–139, May 1992, ISSN: 0020-0190. DOI: 10.1016/0020-0190(92)90136-J.

[103] P. N. Klein, “Computing the Edit-Distance between unrooted ordered trees,” in *Algorithms — ESA’ 98*, Springer Berlin Heidelberg, 1998, pp. 91–102. DOI: 10.1007/3-540-68530-8_8.

[104] S. Dulucq and H. Touzet, “Analysis of tree edit distance algorithms,” in *Combinatorial Pattern Matching*, Springer Berlin Heidelberg, 2003, pp. 83–95. DOI: 10.1007/3-540-44888-8_7.

[105] W. Chen, “New algorithm for ordered tree-to-tree correction problem,” *Journal of algorithms & computational technology*, vol. 40, no. 2, pp. 135–158, Aug. 2001, ISSN: 1748-3018, 0196-6774. DOI: 10.1006/jagm.2001.1170.

[106] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann, “An optimal decomposition algorithm for tree edit distance,” *ACM Transactions on Algorithms*, vol. 6, no. 1, pp. 1–19, Dec. 2010, ISSN: 1549-6325. DOI: 10.1145/1644015.1644017.

[107] K. Bringmann, P. Gawrychowski, S. Mozes, and O. Weimann, “Tree edit distance cannot be computed in strongly subcubic time (unless APSP can),” *ACM Transactions on Algorithms*, vol. 16, no. 4, 2020, ISSN: 1549-6325. DOI: 10.1145/3381878.

[108] T. Flouri, B. Melichar, and J. Janoušek, “Subtree matching by deterministic pushdown automata,” in *2009 International Multiconference on Computer Science and Information Technology*, Mragowo: ieeexplore.ieee.org, Oct. 2009, pp. 659–666, ISBN: 9781424453146. DOI: 10.1109/IMCSIT.2009.5352769.

[109] T. Flouri, B. Melichar, and J. Janoušek, “Aho-Corasick like multiple subtree matching by pushdown automata,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC ’10, Sierre, Switzerland: Association for Computing Machinery, Mar. 2010, pp. 2157–2158, ISBN: 9781605586397. DOI: 10.1145/1774088.1774543.

[110] T. Flouri, C. S. Iliopoulos, J. Janoušek, B. Melichar, and S. P. Pissis, “Tree template matching in ranked ordered trees by pushdown automata,” in *Implementation and Application of Automata*, vol. 17, Springer Berlin Heidelberg, Dec. 2012, pp. 273–281. DOI: 10.1007/978-3-642-22256-6_25.

[111] D. Nowotka and J. Srba, "Height-Deterministic pushdown automata," in *Mathematical Foundations of Computer Science 2007*, Springer Berlin Heidelberg, 2007, pp. 125–134. DOI: 10.1007/978-3-540-74456-6_13.

[112] R. Baeza-Yates and B. Ribeiro-Neto, *Modern information retrieval.* New York: ACM press, 1999, ISBN: 9780201398298.

[113] P. Weiner, "Linear pattern matching algorithms," in *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, IEEE, Oct. 1973, pp. 1–11. DOI: 10.1109/SWAT.1973.13.

[114] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, Oct. 1993, ISSN: 0097-5397. DOI: 10.1137/0222058.

[115] M. Mohri, P. Moreno, and E. Weinstein, "General suffix automaton construction algorithm and space bounds," *Theoretical computer science*, vol. 410, no. 37, pp. 3553–3562, Sep. 2009, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2009.03.034.

[116] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht, "Complete inverted files for efficient text retrieval and analysis," *Journal of the ACM*, vol. 34, no. 3, pp. 578–595, Jul. 1987, ISSN: 0004-5411. DOI: 10.1145/28869.28873.

[117] M. Crochemore and R. Vérin, "On compact directed acyclic word graphs," *Structures in logic and computer science*, 1997. DOI: 10.1007/3-540-63246-8_12.

[118] R. A. Baeza-Yates, "Searching subsequences," *Theoretical computer science*, vol. 78, no. 2, pp. 363–376, Jan. 1991, ISSN: 0304-3975, 1879-2294. DOI: 10.1016/0304-3975(91)90358-9.

[119] M. Crochemore, B. Melichar, and Z. Troníček, "Directed acyclic subsequence graph—overview," *Journal of discrete algorithms*, vol. 1, no. 3, pp. 255–280, Jun. 2003, ISSN: 1570-8667. DOI: 10.1016/S1570-8667(03)00029-7.

[120] M. J. Fischer and M. S. Paterson, "String matching and other products," Tech. Rep., 1974. [Online]. Available: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TM-041.pdf.

[121] C. S. Iliopoulos and M. S. Rahman, "Pattern matching algorithms with don't cares," in *SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007, Proceedings Volume II*, Institute of Computer Science AS CR, Prague, 2007, pp. 116–126.

[122] C. S. Iliopoulos and M. S. Rahman, "Indexing factors with gaps," *Algorithmica. An International Journal in Computer Science*, vol. 55, no. 1, pp. 60–70, Sep. 2009, ISSN: 0178-4617, 1432-0541. DOI: 10.1007/s00453-007-9141-3.

[123] P. Bille and I. L. Gørtz, "Substring range reporting," *Algorithmica. An International Journal in Computer Science*, vol. 69, no. 2, pp. 384–396, Jun. 2014, ISSN: 0178-4617, 1432-0541. DOI: 10.1007/s00453-012-9733-4.

[124] T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-M. Yiu, "Space efficient indexes for string matching with don't cares," in *Algorithms and Computation*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 846–857, ISBN: 9783540771180. DOI: 10.1007/978-3-540-77120-3_73.

[125] P. Bille, I. L. Gørtz, H. W. Vildhøj, and D. K. Wind, "String matching with variable length gaps," *Theoretical computer science*, vol. 443, pp. 25–34, Jul. 2012, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2012.03.029.

[126] M. Lewenstein, "Indexing with gaps," in *String Processing and Information Retrieval*, Springer Berlin Heidelberg, 2011, pp. 135–143. DOI: 10.1007/978-3-642-24583-1_14.

[127] M. Cáceres, S. J. Puglisi, and B. Zhukova, “Fast indexes for gapped pattern matching,” in *SOFSEM 2020: Theory and Practice of Computer Science*, Springer International Publishing, 2020, pp. 493–504. DOI: 10.1007/978-3-030-38919-2_40.

[128] K. Fredriksson and S. Grabowski, “Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance,” *Information retrieval*, vol. 11, no. 4, pp. 335–357, Aug. 2008, ISSN: 1386-4564, 1573-7659. DOI: 10.1007/s10791-008-9054-z.

[129] T. Haapasalo, P. Silvasti, S. Sippu, and E. Soisalon-Soininen, “Online dictionary matching with Variable-Length gaps,” in *Experimental Algorithms*, Springer Berlin Heidelberg, 2011, pp. 76–87. DOI: 10.1007/978-3-642-20662-7_7.

[130] P. Bille, I. L. Gørtz, H. W. Vildhøj, and S. Vind, “String indexing for patterns with wildcards,” *Theory of Computing Systems*, vol. 55, no. 1, pp. 41–60, Jul. 2014, ISSN: 1432-4350, 1433-0490. DOI: 10.1007/s00224-013-9498-4.

[131] J. Bader, S. Gog, and M. Petri, “Practical variable length gap pattern matching,” in *Experimental Algorithms*, Springer International Publishing, 2016, pp. 1–16. DOI: 10.1007/978-3-319-38851-9_1.

[132] P. Peterlongo, J. Allali, and M.-F. Sagot, “Indexing gapped-factors using a tree,” *International Journal of Foundations of Computer Science*, vol. 19, no. 01, pp. 71–87, Feb. 2008, ISSN: 0129-0541. DOI: 10.1142/S0129054108005541.

[133] J. Janoušek, “String suffix automata and subtree pushdown automata,” in *Proceedings of the Prague Stringology Conference 2009*, J. Holub and J. Žďárek, Eds., vol. 2009, Prague, 2009, pp. 160–172, ISBN: 9788001044032. [Online]. Available: https://www.stringology.org/event/2009/p15.html.

[134] R. Goldman and J. Widom, “DataGuides: Enabling query formulation and optimization in semistructured databases,” Tech. Rep., 1997. [Online]. Available: http://ilpubs.stanford.edu:8090/264/.

[135] T. Milo and D. Suciu, “Index structures for path expressions,” in *Database Theory — ICDT’99*, Springer Berlin Heidelberg, 1999, pp. 277–295. DOI: 10.1007/3-540-49257-7_18.

[136] Q. Li and B. Moon, “Indexing and querying XML data for regular path expressions,” in *Proceedings of the 27th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers Inc., 2001, pp. 361–370. [Online]. Available: https://dl.acm.org/doi/10.5555/645927.672035.

[137] C.-W. Chung, J.-K. Min, and K. Shim, “APEX: An adaptive path index for XML data,” in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ser. SIGMOD ’02, Madison, Wisconsin: Association for Computing Machinery, Jun. 2002, pp. 121–132, ISBN: 9781581134971. DOI: 10.1145/564691.564706.

[138] N. Tang, J. X. Yu, M. T. Ozsu, and K. Wong, “Hierarchical indexing approach to support XPath queries,” in *2008 IEEE 24th International Conference on Data Engineering*, ieeexplore.ieee.org, Apr. 2008, pp. 1510–1512. DOI: 10.1109/ICDE.2008.4497606.

[139] R. W. P. Luk, H. V. Leong, T. S. Dillon, A. T. S. Chan, W. B. Croft, and J. Allan, “A survey in indexing and searching XML documents,” *Journal of the American Society for Information Science and Technology*, vol. 53, no. 6, pp. 415–437, 2002, ISSN: 1532-2882, 1532-2890. DOI: 10.1002/asi.10056.

[140] B. Catania, A. Maddalena, and A. Vakali, “XML document indexes: A classification,” *IEEE Internet Computing*, vol. 9, no. 5, pp. 64–71, Sep. 2005, ISSN: 1089-7801, 1941-0131. DOI: 10.1109/MIC.2005.115.

[141] S. A. Mohammad, “Index structures for XML databases,” Ph.D. dissertation, Queen’s University Kingston, Ontario, Canada, 2011.

[142] Q. Zou, S. Liu, and W. W. Chu, "Ctree: A compact tree for indexing XML data," in *Proceedings of the 6th annual ACM international workshop on Web information and data management*, ser. WIDM '04, Washington DC, USA: Association for Computing Machinery, Nov. 2004, pp. 39–46, ISBN: 9781581139785. DOI: 10.1145/1031453.1031462.

[143] P. M. Pettovello and F. Fotouhi, "MTree: An XML XPath graph index," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06, Dijon, France: Association for Computing Machinery, Apr. 2006, pp. 474–481, ISBN: 9781595931085. DOI: 10.1145/1141277.1141389.

[144] H. Jiang, H. Lu, W. Wang, and B. C. Ooi, "XR-tree: Indexing XML data for efficient structural joins," in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, ieeexplore.ieee.org, Mar. 2003, pp. 253–264. DOI: 10.1109/ICDE.2003.1260797.

[145] P. Rao and B. Moon, "PRIX: Indexing and querying XML using prufer sequences," in *Proceedings. 20th International Conference on Data Engineering*, ieeexplore.ieee.org, Apr. 2004, pp. 288–299. DOI: 10.1109/ICDE.2004.1320005.

[146] H. Wang, S. Park, W. Fan, and P. S. Yu, "ViST: A dynamic index method for querying XML data by tree structures," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '03, San Diego, California: Association for Computing Machinery, Jun. 2003, pp. 110–121, ISBN: 9781581136340. DOI: 10.1145/872757.872774.

[147] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe, "Representative objects: Concise representations of semistructured, hierarchical data," in *Proceedings 13th International Conference on Data Engineering*, ieeexplore.ieee.org, Apr. 1997, pp. 79–90. DOI: 10.1109/ICDE.1997.581741.

[148] L. Segoufin and V. Vianu, "Validating streaming XML documents," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ser. PODS '02, Madison, Wisconsin: Association for Computing Machinery, Jun. 2002, pp. 53–64, ISBN: 9781581135077. DOI: 10.1145/543613.543622.

[149] Y. Diao, P. Fischer, M. J. Franklin, and R. To, "YFilter: Efficient and scalable filtering of XML documents," in *Proceedings 18th International Conference on Data Engineering*, ieeexplore.ieee.org, Feb. 2002, pp. 341–342. DOI: 10.1109/ICDE.2002.994748.

[150] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML streams with deterministic automata," in *Database Theory — ICDT 2003*, Springer Berlin Heidelberg, 2003, pp. 173–189. DOI: 10.1007/3-540-36285-1_12.

[151] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML streams with deterministic automata and stream indexes," *ACM Trans. Database Syst.*, vol. 29, no. 4, pp. 752–788, Dec. 2004, ISSN: 0362-5915. DOI: 10.1145/1042046.1042051.

[152] V. Kumar, P. Madhusudan, and M. Viswanathan, "Visibly pushdown automata for streaming XML," in *Proceedings of the 16th international conference on World Wide Web*, ser. WWW '07, Banff, Alberta, Canada: Association for Computing Machinery, May 2007, pp. 1053–1062, ISBN: 9781595936547. DOI: 10.1145/1242572.1242714.

[153] F. Neven, "Automata theory for XML researchers," *SIGMOD Rec.*, vol. 31, no. 3, pp. 39–46, Sep. 2002, ISSN: 0163-5808. DOI: 10.1145/601858.601869.

[154] F. Neven, "Automata, logic, and XML," in *Computer Science Logic*, Springer Berlin Heidelberg, 2002, pp. 2–26. DOI: 10.1007/3-540-45793-3_2.

[155] T. Schwentick, "Automata for XML—A survey," *Journal of Computer and System Sciences*, vol. 73, no. 3, pp. 289–315, May 2007, ISSN: 0022-0000. DOI: 10.1016/j.jcss.2006.10.003.

[156] E. Šestáková, B. Melichar, and J. Janoušek, "Constrained approximate subtree matching by finite automata," in *Proceedings of the Prague Stringology Conference 2018*, J. Holub and J. Žďárek, Eds., Prague, 2018, pp. 79–90, ISBN: 9788001064849. [Online]. Available: `https://www.stringology.org/event/2018/p08.html`.

[157] E. Šestáková, O. Guth, and J. Janoušek, "Automata approach to inexact tree pattern matching using 1-degree edit distance," in *Proceedings of the Prague Stringology Conference 2021*, J. Holub and J. Žďárek, Eds., Czech Technical University in Prague, Czech Republic, 2021, pp. 1–15, ISBN: 9788001068694. [Online]. Available: `https://www.stringology.org/event/2021/p01.html`.

[158] E. Šestáková, O. Guth, and J. Janoušek, "Inexact tree pattern matching with 1-degree edit distance using finite automata," *Discrete applied mathematics*, submitted for publication, ISSN: 0166-218X.

[159] R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms.* Createspace Independent Pub, Oct. 2014, ISBN: 9781502575869.

[160] E. Šestáková and J. Janoušek, "Tree string path subsequences automaton and its use for indexing XML documents," in *Languages, Applications and Technologies*, Springer International Publishing, 2015, pp. 171–181, ISBN: 9783319276533. DOI: `10.1007/978-3-319-27653-3_17`.

[161] E. Šestáková and J. Janoušek, "Indexing XML documents using tree paths automaton," in *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*, ser. OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:14, ISBN: 9783959770569. DOI: `10.4230/OASIcs.SLATE.2017.10`.

[162] E. Šestáková and J. Janoušek, "Automata approach to XML data indexing," *Information*, vol. 9, no. 1, p. 12, Jan. 2018, ISSN: 2078-2489. DOI: `10.3390/info9010012`.

Appendix A

Reviewed publications of the author relevant to the dissertation thesis

This section contains a list of reviewed publications relevant to the dissertation thesis in which I[1] am an author. In most of the publications, I am the only author (apart from my supervisor, Jan Janoušek). However, some of my research has been collaborative. If that is the case, I describe my contribution and the contribution of the remaining authors. These clarifications are intended for the examiners of this dissertation thesis.

The content of Chapter 5 is partially based on the following publications:

1. E. Šestáková, B. Melichar, and J. Janoušek, "Constrained approximate subtree matching by finite automata," in *Proceedings of the Prague Stringology Conference 2018*, J. Holub and J. Žďárek, Eds., Prague, 2018, pp. 79–90, ISBN: 9788001064849. [Online]. Available: `https://www.stringology.org/event/2018/p08.html`

2. E. Šestáková, O. Guth, and J. Janoušek, "Automata approach to inexact tree pattern matching using 1-degree edit distance," in *Proceedings of the Prague Stringology Conference 2021*, J. Holub and J. Žďárek, Eds., Czech Technical University in Prague, Czech Republic, 2021, pp. 1–15, ISBN: 9788001068694. [Online]. Available: `https://www.stringology.org/event/2021/p01.html`

3. E. Šestáková, O. Guth, and J. Janoušek, "Inexact tree pattern matching with 1-degree edit distance using finite automata," *Discrete applied mathematics*, submitted for publication, ISSN: 0166-218X

In this dissertation thesis, I explain the results in more detail than it is done in the above-mentioned publications and describe the relationship with the SNINWE classification proposed in Chapter 3. I also present slightly different variants of some algorithms as discussed below.

I co-authored the first publication mentioned above with Bořivoj Melichar. He was the one who came up with an idea to explore the string automata approach to inexact tree pattern matching. Melichar also supervised my work, gave me feedback, and read my drafts.

I co-authored the second and third publication with Ondřej Guth. He is the main author behind the idea of dynamic programming algorithm for the (simple) 1-degree edit distance. However, in this dissertation thesis, I present a slightly different version which has better space complexity and it is, in my opinion, conceptually simpler.

[1] The personal pronoun "I" instead of "we" is used in this part for the purpose of describing the contributions of the author of this dissertation thesis.

The content of Chapter 6 is partially based on the following publications:

1. E. Šestáková and J. Janoušek, "Tree string path subsequences automaton and its use for indexing XML documents," in *Languages, Applications and Technologies*, Springer International Publishing, 2015, pp. 171–181, ISBN: 9783319276533. DOI: 10.1007/978-3-319-27653-3_17

2. E. Šestáková and J. Janoušek, "Indexing XML documents using tree paths automaton," in *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*, ser. OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:14, ISBN: 9783959770569. DOI: 10.4230/OASIcs.SLATE.2017.10

3. E. Šestáková and J. Janoušek, "Automata approach to XML data indexing," *Information*, vol. 9, no. 1, p. 12, Jan. 2018, ISSN: 2078-2489. DOI: 10.3390/info9010012

   This article has been cited in

   - P. Sijin and H. N. Champa, "Dynamic document localization for efficient mining," in *Sentimental Analysis and Deep Learning*, Springer Singapore, 2022, pp. 15–29. DOI: 10.1007/978-981-16-5157-1_2
   - R. Chen, Z. Wang, H. Su, *et al.*, "Parallel XPath query based on cost optimization," *The Journal of supercomputing*, vol. 78, no. 4, pp. 5420–5449, Mar. 2022, ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-021-04074-y
   - P. Sijin and H. N. Champa, "Data conceptualization for semantic search diversification over XML data," in *2021 2nd International Conference on Smart Electronics and Communication (ICOSEC)*, Oct. 2021, pp. 875–882. DOI: 10.1109/ICOSEC51865.2021.9591868

In the above publications, the methods are specifically tailored to XML. In this dissertation thesis, I present the results more generally considering arbitrary ordered labeled trees. The explanation of individual methods is also much more detailed, and the methods are presented in the context of the SNINWE classification of the tree pattern matching problems proposed in Chapter 3. Moreover, the path automaton described in Section 6.4 is only described in this dissertation thesis and not in the above-mentioned publications.

Appendix B

Remaining publications of the author relevant to the dissertation thesis

- E. Šestáková, *Automaty a gramatiky: sbírka řešených příkladů*, cs. Czech Technical University in Prague, 2017, ISBN: 9788001063064
- E. Šestáková, *Automata and grammars: A collection of exercises and solutions*. Czech Technical University in Prague, 2018, ISBN: 9788001064627
- E. Šestáková and O. Guth, *Lecture notes on automata, languages, and grammars*, cs. 2021
- E. Šestáková, "Indexing XML documents and tree data structures," Czech Technical University in Prague, Tech. Rep., 2017
- E. Šestáková, "Automata approach to processing tree data structures," Czech Technical University in Prague, Tech. Rep., 2017

Appendix C

Remaining publications of the author

- C. Piech, L. Yan, L. Einstein, *et al.*, "Co-Teaching computer science across borders: Human-Centric learning at scale," in *Proceedings of the Seventh ACM Conference on Learning @ Scale*, ser. L@S '20, Virtual Event, USA: Association for Computing Machinery, Aug. 2020, pp. 103–113, ISBN: 9781450379519. DOI: 10.1145/3386527.3405915